Carderock Division
Naval Surface Warfare Center
Bethesda, Maryland 20817-5700

NSWCCD-26-TR-1998/xx October 1998

Total Ship Systems Directorate
Research and Development Report

Leading Edge Advanced Prototyping For Ships (LEAPS):

LEAPS User's Guide Version 2.0

by
Richard T. Van Eseltine and Robert Ames

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

Background

In 1996, an innovation team was formed at the Carderock Division, Naval Surface Warfare Center (NSWCCD) to investigate the issues of virtual prototyping and modeling and simulation. This team became known as the **LEAPS** (Leading Edge Advanced Prototyping for Ships) team. The efforts of this team led to the development of an architecture that facilitated an integrated virtual prototyping process for ship concept assessments. The vision for the virtual prototyping process for ships is similar to that developed for tank concept assessments by the Army Tank Automotive Command (TACOM). TACOM won a Presidential Quality Award for that development. The current scope of the LEAPS development effort encompasses the first five steps of the TACOM process:

1. mission requirements identification,
2. concepts development;
3. performance modeling,
4. warfare analysis; and,
5. to a lesser extent, detailed design.

About the LEAPS Architecture

The LEAPS architecture is a framework that can support conceptual surface ship and submarine design and analysis through DoD acquisition Milestone 1, i.e., early stage ship design. Due to the complexity and diversity of naval ship design and analysis, the LEAPS architecture takes a "meta model" approach to product model development. This general engineering approach has some classes that may be considered specific to naval ship design, but most classes would be applicable to development any engineering product.

To understand the LEAPS architecture it is necessary to have a taxonomy for concepts such as Application Programming Interface, Product Model, Meta Model, and other terms used today in discussing integrated environments and their computational framework. The intent is not to find agreement with the terminology, only to give it context within the LEAPS architecture.

The **LEAPS MetaModel (LEAPS/MM**) is a set of entities that describe representations and methods that can be used in defining a smart product model. In particular, these entities allow complex engineering representations such as ship modeling. For example, there are entities for geometric representation, performance behaviors, component and subsystem definition, and processes such as Studies. While this metamodel was based on ship design and analysis requirements, it is

general enough that it could be used in the development of almost any product model.

The **LEAPS Application Programming Interface** (**LEAPS/API**) is a set of C++ classes available for application programming that implement the LEAPS/MM. The LEAPS/API is a library written using ISO standard C++ and is accessible in both static and dynamic link libraries. This API is used to write translators to retrieve and store data associated with the **LEAPS Product Model (LEAPS/PM).**

The **LEAPS Product Model (LEAPS/PM)** is the object oriented schema or product characterization of the product, or ship, as defined by individual IPT's. The product model for a submarine would be different than a monohull surface combatant. The LEAPS/MM supports the development of either product model.

The **LEAPS Data Base (LEAPS/DB)** is the persistent store for any LEAPS/PM. A LEAPS/DB can be shared by multiple distributed computers and operating systems: UNIX, Windows 95/NT, and Macintosh.

LEAPS Applications (LEAPS/APP) are individual applications that communicate with the LEAPS/DB through the LEAPS/API. In many cases these individual applications are wrappers to legacy codes. These wrappers allow both modern and legacy programs to create and analyze product model data. There are some applications currently in development that will provide a common interface to the LEAPS/DB that will deal with executive control and product visualization.

The **LEAPS STEP (LEAPS/STEP)** is an ISO STEP translation service for exchange of geometry data through Part21 files. It has been demonstrated but is not available at this time.

The **LEAPS CORBA (LEAPS/CORBA)** is a CORBA interface to the LEAPS/API. Current efforts to incorporate a CORBA interface to the LEAPS/API are underway. This capability has been demonstrated but is not available at this time.

System Requirements for LEAPS/API

LEAPS/API is written using ISO standard C++. It currently being development under Microsoft Windows (Win32) using Microsoft Visual C++. At the time this documentation was written, this compiler supported all of the ISO standard features used by LEAPS. When other platforms bring their compilers into full compliance with the ISO standard, distribution of LEAPS on these platforms will be supported.

Installation

LEAPS is not an application. As such, there is no installation utility. It is recommended, however, that you copy the files and their respective

directories to a location on your computer where you would expect to be doing development.

References, Help, Technical Support

LEAPS reference material is available through the web site at http://ocean.dt.navy.mil/leaps

This web site will contain the latest versions of libraries, documentation and sample code. Technical support is available for funded projects and collaborative initiatives. Contact Myles Hurwitz, mhurwitz@dt.navy.mil, (301) 227-1927, if you would like information on how your organization can participate in the LEAPS environment.

OVERVIEW OF LEAPS

Overview of Primary Classes

Not yet documented

Overview of Geometry Object Structure (GOBS) Classes

The LEAPS GOBS classes allows CAD system geometry and attributes to be presented to engineering modelers and analysts in a form which allows for convenient discretization according to the requirements of their models. The GOBS model purports that geometric product model data is defined and represented as 'views' of geometric objects. The word "view" is in quotes because it is actually an object that appears as geometry. This is not to say that GOBS does not allow geometric objects to represent geometric product model data only that another more powerful approach is available. This is contrary to most CAD representations where the geometry defines the view and the object simultaneously. In addition, GOBS contains connection entities that define common boundaries between objects like the intersection at a deck edge and the hull. The boundary of the deck knows where it is located on the hull and visa versa.

A discussion of some of the GOBS objects follows. It is important to understand that due to space constraints not all GOBS objects are discussed nor are the various methods available to applications through the LEAPS/API.

To understand one aspect of this new geometry topology, an example of three compartments within a ship, depicted in Figure 1 is illustrated. This three compartment case, while simplistic in appearance, actually poses a number of challenges to product modeling. Consider the "knowledge" that must exist at transverse bulkhead 2, (Trans-2). This bulkhead plays a number of roles one of which is the boundary of three compartments.

The bulkhead is connected to the hull, port and starboard, the longitudinal, and both decks above and below. In addition, there are locations on this bulkhead that may be of interest to analysts such as the corner points at intersections with other surfaces (longitudinal, hull, deck, etc.). This bulkhead also plays a role as a boundary, or Face, of each individual compartment. These boundaries can be described as "views" of the bulkhead as seen by each compartment and unique to each compartment. Consider also that the object Trans-2 may play a role, or roles, in many other "views" such as a watertight bulkhead bounding a zone on the ship.

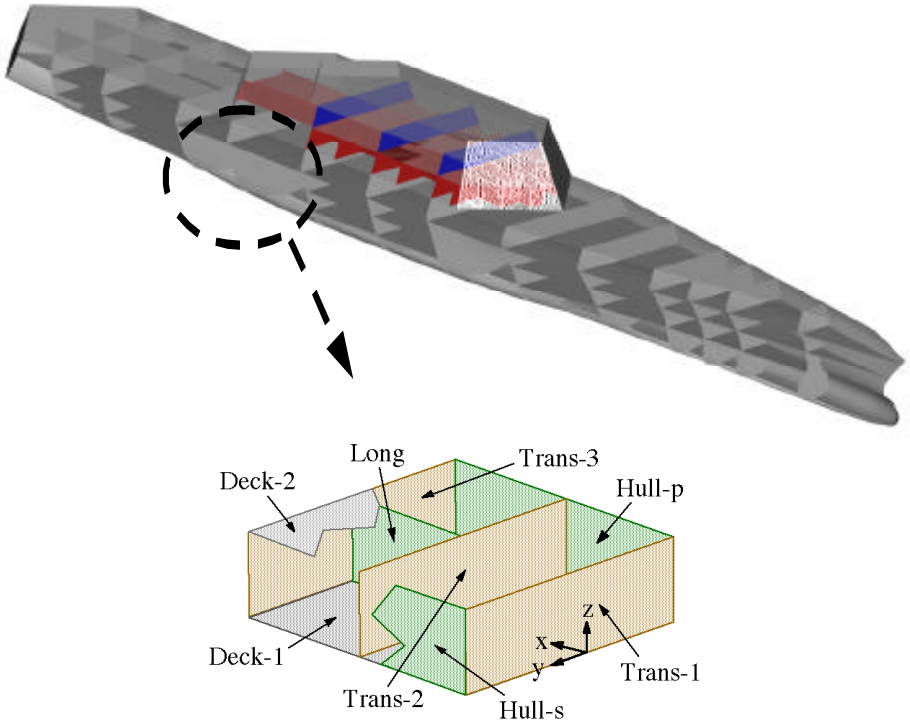

Figure 1 - Three Compartment Test Case

Product Model Views

In GOBS "views" of product model data are actually objects that compose existing geometry into unique physical objects. Similarly there are views that associate physical objects into like groupings. Views that create physical objects from geometry elements are called *Topological Views.* Views that associate *Topological Views* into common groups are called *Common Views*.

Topological Views

The term *Topological View* is foreign to most familiar with geometric modeling. Its best to think of them as traditional surfaces, trimmed surfaces, and Brep solids, with additional capability. The construction of *Topological Views* allows for member shape objects, like surfaces and solids, used in the creation of a *Topological View,* to also play a role as geometric members in others *Topological Views*.

Common Views

Common Views do not have any spatial constraints, unlike *Topological Views*, they are simply a logical grouping of *Topological Views. Common Views* can also have other *Common Views* as members. *Common Views* are the primary vehicle by which domain analyst or designers will view or interrogate the product model. One example of a *Common View* could be "Habitability Spaces on Deck 3". Another *Common View* called "Ship

Habitability Spaces" could contain the *Common View* "Habitability Spaces on Deck 3" as a member. Similar uses of *Common Views* could include "Exterior Surfaces", "Compartments", "Machinery Spaces", or "Mast".

Shape Objects

Some distinctions should be made of the differences between GOBS shape objects and what can be considered typical geometric entities in applications that use and compose geometry such as CAD systems. In GOBS, geometry (*Topological Views*) is the association of shape and *Properties*. Current shape objects are *Surfaces*, trimmed surfaces (*Faces*), and manifold Brep *Solids*.

One major difference in GOBS modeling is the representation of *Faces*. Currently CAD systems today consider a *Face* to be composed of a single *Surface* bounded by a single outer boundary and any number of inner boundaries. The typical CAD model does not allow the underlying *Surface* to be used in the construction of any other *Face.* It requires that a copy of that *Surface* be made. GOBS, on the other hand, allows for a single S*urface* to be used in the construction of any number of *Faces*; where the *Face* object contains reference to one *Surface*, one outer *EdgeLoop*, and any number of inner *EdgeLoops*. This concept is illustrated more clearly in Figure 2 where the deck on a ship is shown highlighting three *Faces* used as compartment boundaries. All three *Faces* share a common deck *Surface* and are defined by a selection of *Edges* that compose a bounded *EdgeLoop.*

Because *Surfaces, Faces*, and *Solids* are shape objects they have no *Properties*. In GOBS the *Topological View* class associates member shape objects with physical characteristics or *Properties* and can be thought of as a geometric component, or part. The *Topological View* has *Properties* of a physical or performance nature, where the underlying *Surface, Face*, or *Solid* object, is simply providing information on its shape. As *Topological Views* are composed, the grouping into *Common Views* is the next natural step.

In Figure 2, *Topological Views* of regions on a deck are illustrated. In this case they appear as "Comp 1 Deck", "Comp 2 Deck", and "Comp 3 Deck". Each *Topological View* use *Faces* ("FA1", "FA2", "FA3") to define their shape within compartments (*Common Views* named "Compartment 1", "Compartment 2", "Compartment 3"). Similarly these *Common View* compartments are also shown as members of a single *Common View* defining a zone ("Zone 1"). In summary, this example demonstrates how space with each compartment derives their shape from *Faces*. These *Faces* are defined as simple boundaries (*EdgeLoops)* on single underlying geometric element which is a *Surface* (Deck 1) defining the entire deck shape. These spaces are then associated in a *Common View* to support design domain knowledge.

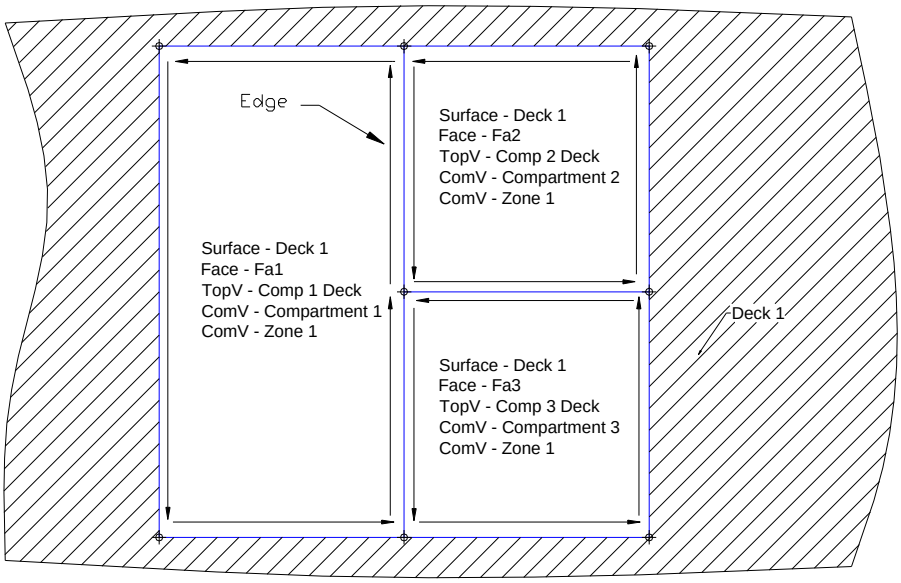

Figure 2 - Three Compartments on Deck 1

Again, GOBS takes the position that topology is a view of space not the space itself. An office room, ship's compartments, or other like space, can be viewed as the collection of faces that make the walls, floor, and ceiling. To the occupant the wall of the room extends to the intersection of other walls, ceiling, and floor. The wall, however, may be defined as the space bounded by the outside walls of a building. Thus, the office room could be represented as a list of connected faces where the view of the wall is the region of the larger wall surface with local boundaries applied.

Another fundamental feature of GOBS is the CoEdge object. The CoEdge provides a unique role in the discretization of the geometry for analysis. Essentially, a CoEdge knows all edges located on each surface and declares them to be equivalent in 3 space, see Figure 3. It also contains an n-dimensional spline function that maps the parameterization of each edge into a single function. This allows for the continuity of points along one surface to migrate to another surface without having to perform closest point approximations.

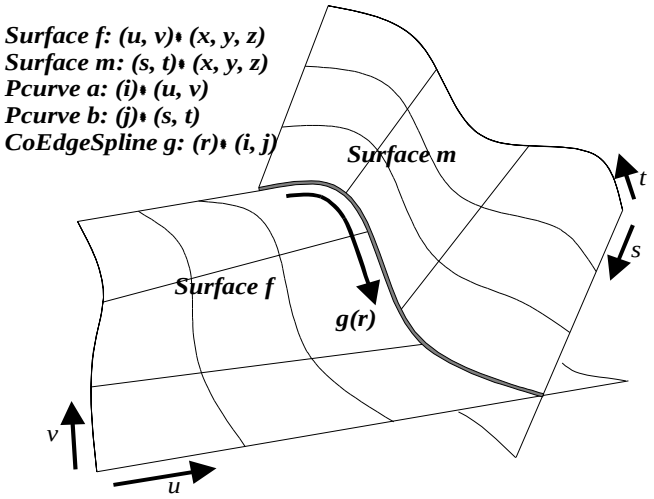

Figure 3 – CoEdge Spline Dependencies

With this topology the ability to traverse boundaries, both logically, explicitly, and with information on the relationship of surface parameter space affords many advantages. Clearly the ability to grid or mesh across trim surface boundaries with node continuity is the most obvious. With the GOBS objects available as a CORBA service to legacy applications, the communication of a single product model geometry in multiple views provides a efficient and effective means for multidiscipline analysis.

Overview of Utility Classes

Not yet documented

GETTING STARTED

Defining and Designing Your Product Model

Defining your product model is a complex task. There are many references to product model design and the subject is too broad to cover here. There has been some product model development done in support of LEAPS that pertains to ship design and analysis, but this is an ongoing effort. Likewise, STEP application protocols for ships have been developed by and for shipyards that has some overlap with concept design and analysis. This also is an evolving standard.

Figure 4 shows an example of a ship concepts analysis data flow process that is driven by an Integrated Process Team (**IPT**) It was created as part of the initial LEAPS effort. Such a model helps to define a use case for your product model.

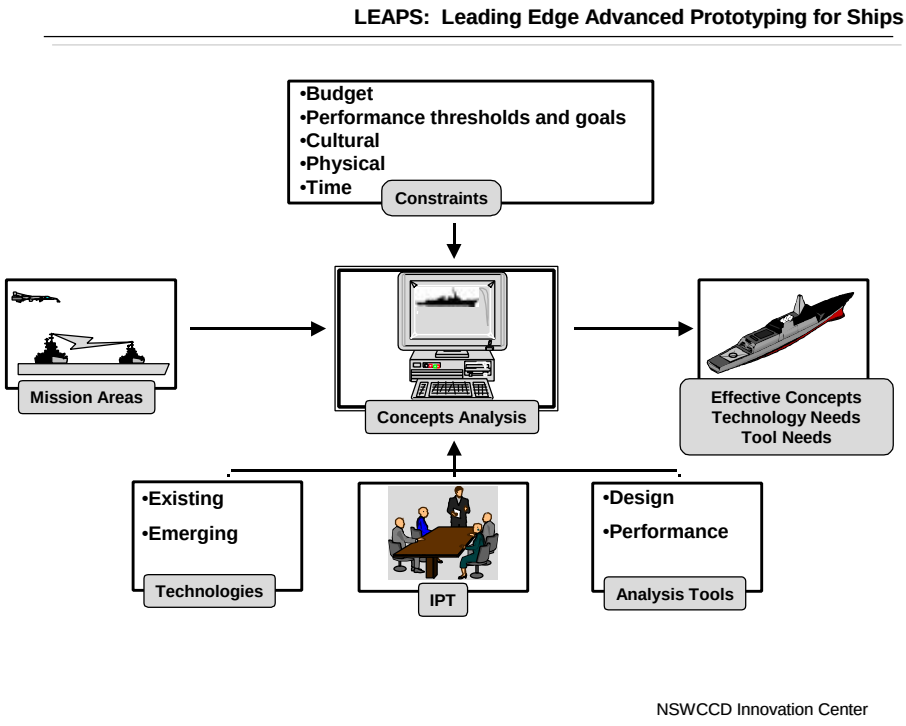

Figure 4 - LEAPS-supported Concept Development Process

The IPT is responsible to define and design the LEAPS/PM. The IPT must determine what attributes are needed by the study and define these attributes. The IPT must also determine where the attributes belong in the Leaps/MM and specify the analyst responsible for providing the data for the attribute. Figure 5 shows an example of a part of a LEAPS/PM that was defined by the LEAPS team.

Instance	Class	Type	Parent	Owner
hull_principal_dimensions	Lps::PropertyGroupPtr		hull_form	
lbp	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
beam	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
draft	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
depth_sta_0	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
depth_sta_3	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
depth_sta_10	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
depth_sta_20	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
prismatic_coef	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
max_section_coef	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
waterplane_coef	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
lcb_lbp	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
lcf_lbp	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
half_siding_width	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
bot_rake	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
main_deck_ht	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
raised_deck_ht	Lps::PropertyPtr	Lps::RealScalarPtr	hull_principal_dimensions	
raised_deck_limits_array	Lps::PropertyPtr	Lps::RealSTLVectorPtr	hull_principal_dimensions	

Figure 5 - Example of LEAPS Product Model

The analyst is responsible for the translator that retrieves the attributes he needs to develop his input for his analysis program. Thus, the analyst must determine what attributes are needed from the LEAPS/PM to develop his input for his analysis program. The analyst is also responsible for the translator that stores the attributes he is responsible for in the LEAPS/PM. The analyst must insure that attributes he provides are being used appropriately by other members of the IPT.

Compiling Your Application Using the LEAPS API

Compiling your application using the LEAPS class library is dependent on the platform (i.e. operation system), the compiler, and linker being used. The compiler must be fully compliant with the ISO C++ standard. Currently, LEAPS libraries are only available for Windows NT 4.0.

There are three steps needed to ensure the proper generation of an executable.

1. The include path must be setup properly to ensure that directory references to all the include files needed by the LEAPS API are established. Using Visual C++ 5.0, this is done with the /I option for the compile command. For example, if the include path for LEAPS is L:\leaps\include, then **/I "L:\leaps\include"** would be added as an option to the compile command.

2. Like the include path in step 1, the library path that contains the LEAPS library (this includes the DTNURBS library) must also be located by the linker. Using Visual C++ 5.0, this is done with the /libpath option for the link command. For example, if the library path

for LEAPS is L:\leaps\libraries, then **/libpath:"L:\leaps\libraries"** would be added as an option to the link command.

3. The appropriate LEAPS and DTNURBS libraries must also be included in the link command. For example using Visual C++ 5.0 in a console application, LeapsV2StaticRelease.lib and dtnurbsV35StaticRelease.lib would be added to the link command. The appropriate libraries (dtnurbsV35DLLRelease.lib, LeapsV2DLLRelease.lib) would be substituted for WIN32 DDL based applications.

   If the DLL version of the LEAPS library is used, application programs should be compiled with the proper multithreaded DLLs and __DLLIMPORT should be defined.

Creating a LEAPS Database

LEAPS/API stores all persistent data in a database. This database is a directory on the file system where the LEAPS/PM data will be stored. The user simply creates a directory on the file system. The user has the option of referencing this database through a user defined environment variable or specifying the directory path explicitly. Once the directory is created, modification of this directory should only occur through the LEAPS API. If modification of this directory occurs by other means, it could corrupt the database.

MANAGEMENT OF LEAPS OBJECTS

Managing a LEAPS Database

A LEAPS Factory is created to manage a LEAPS database. To create a Factory to manage a database whose directory is SampleLeapsDB, code such as:

```
Lps::Factory leapsDB;
leapsDB = Lps::Factory::create ("SampleLeapsDB");
```

is used.

Managing Studies

A LEAPS Factory manages LEAPS Study objects that are in a LEAPS database. The Factory has member methods that provide various functions that involve Study objects. These member methods can:

- Create a Study object,
- Retrieve a Study object,
- Destroy a Study object,
- Determine the existence of a Study object, and
- List Study objects that are managed by the Factory.

Creating a Study

If leapsDB is a factory that has been created to manage a LEAPS database, a program can create a Study with the following code.

```
Lps::StudyPtr cvxStudy;
cvxStudy = leapsDB->createStudy ("cvxStudy", 1);
```

Retrieving a Study

If leapsDB is a factory that has been created to manage a LEAPS database, a program can retrieve a Study with the following code.

```
Lps::StudyPtr cvxStudy;
cvxStudy = leapsDB->getStudy ("cvxStudy", 1);
```

Destroying a Study

If leapsDB is a factory that has been created to manage a LEAPS database, a program can destroy a Study with the following code.

```
leapsDB->destroyStudy ("cvxStudy", 1);
```

Determining the Existence of a Study

If leapsDB is a factory that has been created to manage a LEAPS database, the following code determines if a study exists.

```
If (leapsDB->doesStudyExist ("cvxStudy", 1)
  std::cout << "Study Exists" << std::endl;
else
  std::cout << "Study does NOT Exist" << std::endl;
```

Listing the Studies Managed by a Factory

If leapsDB is a factory that has been created to manage a LEAPS database, a program can obtain a list of the studies unique identifiers that are contained within the database with the following code.

```
Lps::UniqueIdList uidList = leapsDB->getUidsOfStudies ();
```

or a list of names and versions of the studies with

```
Lps::NameVersionPairList nvList;
nvList = leapsDB->getNameVersionPairsOfStudies ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing Concepts

A LEAPS Study object manages LEAPS Concept objects that are contained by it. The Study has member methods that provide various functions that involve Concept objects. These member methods can:

- Create a Concept object,
- Retrieve a Concept object,
- Destroy a Concept object,
- Determine the existence of a Concept object,
- List Concept objects that are contained within the Study, and
- Retrieve a Concept's structure (geometry).

Creating a Concept

If cvxStudy is a Study object that has been retrieved, a program can create a Concept with the following code.

```
Lps::ConceptPtr cvx;
cvx = cvxStudy->createConcept ("cvx", 1);
```

Retrieving a Concept

If cvxStudy is a Study object that has been retrieved, a program can retrieve a Concept with the following code.

```
Lps::ConceptPtr cvx;
cvx = cvxStudy->getConcept ("cvx", 1);
```

Destroying a Concept

If cvxStudy is a Study object that has been retrieved, a program can destroy a Concept with the following code.

```
cvxStudy->destroyConcept ("cvx", 1);
```

Determining the Existence of a Concept

If cvxStudy is a Study object that has been retrieved, the following code determines if a Concept exists.

```
If (cvxStudy->doesConceptExist ("cvx", 1)
  std::cout << "Concept Exists" << std::endl;
else
  std::cout << "Concept does NOT Exist" << std::endl;
```

Listing the Concepts Contained in a Study

If cvxStudy is a Study object that has been retrieved, a program can obtain a list of the concepts unique identifiers that are contained within the Study with the following code.

```
Lps::UniqueIdList uidList = cvxStudy->getUidsOfConcepts ();
```

or a list of names and versions of the concepts with

```
Lps::NameVersionPairList nvList;
nvList = cvxStudy->getNameVersionPairsOfConcepts ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Retrieving a Concept's Structure

If cvx is a Concept object that has been retrieved, a program can obtain the concept's structure with the following code.

```
Lps::StructurePtr cvxStructure;
cvxStructure = cvx->getConceptStructure ();
```

Managing Scenarios

A LEAPS Study object manages LEAPS Scenario objects that are contained by it. The Study has member methods that provide various functions that involve Scenario objects. These member methods can:

- Create a Scenario object,
- Retrieve a Scenario object,
- Destroy a Scenario object,
- Determine the existence of a Scenario object, and
- List Scenario objects that are contained within the Study.

Creating a Scenario

If cvxStudy is a Study object that has been retrieved, a program can create a Scenario with the following code.

```
Lps::ScenarioPtr situationA;
situationA = cvxStudy->createScenario ("situationA", 1);
```

Retrieving a Scenario

If cvxStudy is a Study object that has been retrieved, a program can retrieve a Scenario with the following code.

```
Lps::ScenarioPtr situationA;
situationA = cvxStudy->getScenario ("situationA", 1);
```

Destroying a Scenario

If cvxStudy is a Study object that has been retrieved, a program can destroy a Scenario with the following code.

```
cvxStudy->destroyScenario ("situationA", 1);
```

Determining the Existence of a Scenario

If cvxStudy is a Study object that has been retrieved, the following code determines if a Scenario exists.

```
If (cvxStudy->doesScenarioExist ("situationA", 1)
  std::cout << "Scenario Exists" << std::endl;
else
  std::cout << "Scenario does NOT Exist" << std::endl;
```

Listing the Scenarios Contained in a Study

If cvxStudy is a Study object that has been retrieved, a program can obtain a list of the Scenarios unique identifiers that are contained within the Study with the following code.

```
Lps::UniqueIdList uidList = cvxStudy->getUidsOfScenarios ();
```

or a list of names and versions of the Scenarios with

```
Lps::NameVersionPairList nvList;
nvList = cvxStudy->getNameVersionPairsOfScenarios ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing Components

A LEAPS Concept object is composed of LEAPS Component objects. The Concept has member methods that provide various functions that involve the management of Component objects. These member methods can:

- Create a Component object,
- Retrieve a Component object,
- Destroy a Component object,
- Determine the existence of a Component object,
- List Component objects that are contained within the Concept, and
- Retrieve a Component's structure (geometry).

Creating a Component

If cvxConcept is a Concept object that has been retrieved, a program can create a Component with the following code.

```
Lps::ComponentPtr pump;
pump = cvxConcept->createComponent ("pump", 1);
```

Retrieving a Component

If cvxConcept is a Concept object that has been retrieved, a program can retrieve a Component with the following code.

```
Lps::ComponentPtr pump;
pump = cvxConcept->getComponent ("pump", 1);
```

Destroying a Component

If cvxConcept is a Concept object that has been retrieved, a program can destroy a Component with the following code.

```
cvxConcept->destroyComponent ("pump", 1);
```

Determining the Existence of a Component

If cvxConcept is a Concept object that has been retrieved, the following code determines if a Component exists.

```
If (cvxConcept->doesComponentExist ("pump", 1)
  std::cout << "Component Exists" << std::endl;
else
  std::cout << "Component does NOT Exist" << std::endl;
```

Listing the Components Contained in a Concept

If cvxConcept is a Concept object that has been retrieved, a program can obtain a list of the Components unique identifiers that are contained within the Concept with the following code.

```
Lps::UniqueIdList uidList = cvxConcept->getUidsOfComponents ();
```

or a list of names and versions of the Components with

```
Lps::NameVersionPairList nvList;
nvList = cvxConcept->getNameVersionPairsOfComponents ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Retrieving a Component's Structure

If pump is a Component object that has been retrieved, a program can obtain the component's structure with the following code.

```
Lps::StructurePtr pumpStructure;
pumpStructure = pump->getComponentStructure ();
```

Managing Systems

A LEAPS Concept object is composed of LEAPS System objects. The Concept has member methods that provide various functions that involve the management of System objects. These member methods can:

- Create a System object,
- Retrieve a System object,
- Destroy a System object,
- Determine the existence of a System object, and
- List System objects that are contained within the Concept.

Creating a System

If cvxConcept is a Concept object that has been retrieved, a program can create a System with the following code.

```
Lps::SystemPtr fireMain;
fireMain = cvxConcept->createSystem ("fireMain", 1);
```

Retrieving a System

If cvxConcept is a Concept object that has been retrieved, a program can retrieve a System with the following code.

```
Lps::SystemPtr fireMain;
fireMain = cvxConcept->getSystem ("fireMain", 1);
```

Destroying a System

If cvxConcept is a Concept object that has been retrieved, a program can destroy a System with the following code.

```
cvxConcept->destroySystem ("fireMain", 1);
```

Determining the Existence of a System

If cvxConcept is a Concept object that has been retrieved, the following code determines if a System exists.

```
If (cvxConcept->doesSystemExist ("fireMain", 1)
  std::cout << "System Exists" << std::endl;
else
```

```
std::cout << "System does NOT Exist" << std::endl;
```

Listing the Systems Contained in a Concept

If cvxConcept is a Concept object that has been retrieved, a program can obtain a list of the Systems unique identifiers that are contained within the Concept with the following code.

```
Lps::UniqueIdList uidList = cvxConcept->getUidsOfSystems ();
```

or a list of names and versions of the Systems with

```
Lps::NameVersionPairList nvList;
nvList = cvxConcept->getNameVersionPairsOfSystems ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing Connections

A LEAPS Concept object is composed of LEAPS Connection objects. The Concept has member methods that provide various functions that involve the management of Connection objects. These member methods can:

- Create a Connection object,
- Retrieve a Connection object,
- Destroy a Connection object,
- Determine the existence of a Connection object, and
- List Connection objects that are contained within the Concept.

Creating a Connection

If cvxConcept is a Concept object that has been retrieved, a program can create a Connection with the following code.

```
Lps::ConnectionPtr fireMainConnection;
fireMainConnection = cvxConcept->createConnection
("fireMainConnection", 1);
```

Retrieving a Connection

If cvxConcept is a Concept object that has been retrieved, a program can retrieve a Connection with the following code.

```
Lps::ConnectionPtr fireMainConnection;
fireMainConnection = cvxConcept->getConnection
("fireMainConnection", 1);
```

Destroying a Connection

If cvxConcept is a Concept object that has been retrieved, a program can destroy a Connection with the following code.

```
cvxConcept->destroyConnection ("fireMainConnection", 1);
```

Determining the Existence of a Connection

If cvxConcept is a Concept object that has been retrieved, the following code determines if a Connection exists.

```
If (cvxConcept->doesConnectionExist ("fireMainConnection", 1)
  std::cout << "Connection Exists" << std::endl;
else
  std::cout << "Connection does NOT Exist" << std::endl;
```

Listing the Connections Contained in a Concept

If cvxConcept is a Concept object that has been retrieved, a program can obtain a list of the Connections unique identifiers that are contained within the Concept with the following code.

```
Lps::UniqueIdList uidList = cvxConcept->getUidsOfConnections ();
```

or a list of names and versions of the Connections with

```
Lps::NameVersionPairList nvList;
nvList = cvxConcept->getNameVersionPairsOfConnections ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing Diagrams

A LEAPS Concept object is composed of LEAPS Diagram objects. The Concept has member methods that provide various functions that involve the management of Diagram objects. These member methods can:

- Create a Diagram object,
- Retrieve a Diagram object,
- Destroy a Diagram object,
- Determine the existence of a Diagram object, and
- List Diagram objects that are contained within the Concept.

Creating a Diagram

If cvxConcept is a Concept object that has been retrieved, a program can create a Diagram with the following code.

```
Lps::DiagramPtr fireMainDiagram;
fireMainDiagram = cvxConcept->createDiagram ("fireMainDiagram",
1);
```

Retrieving a Diagram

If cvxConcept is a Concept object that has been retrieved, a program can retrieve a Diagram with the following code.

```
Lps::DiagramPtr fireMainDiagram;
fireMainDiagram = cvxConcept->getDiagram ("fireMainDiagram", 1);
```

Destroying a Diagram

If cvxConcept is a Concept object that has been retrieved, a program can destroy a Diagram with the following code.

```
cvxConcept->destroyDiagram ("fireMainDiagram", 1);
```

Determining the Existence of a Diagram

If cvxConcept is a Concept object that has been retrieved, the following code determines if a Diagram exists.

```
If (cvxConcept->doesDiagramExist ("fireMainDiagram", 1)
  std::cout << "Diagram Exists" << std::endl;
else
  std::cout << "Diagram does NOT Exist" << std::endl;
```

Listing the Diagrams Contained in a Concept

If cvxConcept is a Concept object that has been retrieved, a program can obtain a list of the Diagrams unique identifiers that are contained within the Concept with the following code.

```
Lps::UniqueIdList uidList = cvxConcept->getUidsOfDiagrams ();
```

or a list of names and versions of the Diagrams with

```
Lps::NameVersionPairList nvList;
nvList = cvxConcept->getNameVersionPairsOfDiagrams ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing LEAPS Geometry Objects

A LEAPS Structure Object is part of a LEAPS Concept object or a LEAPS Component object.. The Structure object manages all LEAPS geometry objects. A Structure object has member methods that provide various functions that involve the management of LEAPS geometry objects. These member methods can:

- Create LEAPS geometry objects,
- Retrieve LEAPS geometry objects,
- Destroy LEAPS geometry objects,
- Determine the existence of LEAPS geometry objects, and
- List LEAPS geometry objects that are contained within the Structure.

Creating a Surface

A LEAPS surface is defined by a non-uniform rational B-spline (NURBS). The spline is defined by a STL vector of SplineDomainVaribles and a STL vector of SplineRangeVariables. If the spline is rational a STL vector of weights is also part of the spline. A SplineDomainVariable is defined with a label, the order, and the knots associated with the domain variable. A SplineRangeVariable is defined by a label and the control points associated with the range variable. There a two SplineDomainVariables that are the u and v parametric variables of the spline and three SplineRangeVariables that are the x, y, and z control points. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a Surface with the following code.

```
// define the order of the domain variable
Lps::Uint32 order4 = 4;

// define array of knots
Lps::Real64 knotArray[9] = {0, 0, 0, 0, 0.5, 1, 1, 1, 1};
std::vector <Lps::Real64> knots;
for (int i = 0 ; i < 9 ; ++i)
  knots.push_back (knotArray[i]);

// create u domain variable
Lps::SplineDomainVariable uEntry;
uEntry.create (knots, order4, "u");

// create v domain variable using the same knots and order
Lps::SplineDomainVariable vEntry;
vEntry.create (knots, order4, "v");

// create the STL vector of domain variables
std::vector<Lps::SplineDomainVariable> domainData;
domainData.push_back (uEntry);
domainData.push_back (vEntry);

// create STL vector of weights
std::vector <Lps::Real64> weights;
for (i = 0 ; i < 25 ; ++i)
  weights.push_back (1.0);

// create STL vector of x control points
Lps::Real64 xCtlPtArray[] = {0,5,10,15,20,0,5,10,15,20,0,5,10,15,
                             20, 0,5,10,15,20,0,5,10,15,20};
std::vector <Lps::Real64> xCtlPts;
for (i = 0 ; i < 25 ; ++i)
  xCtlPts.push_back (xCtlPtArray[i]);

// create x SplineRangeVariable
Lps::SplineRangeVariable xEntry;
xEntry.create (xCtlPts, "x");

// create STL vector of y control points
Lps::Real64 yCtlPtArray[] = {-10,-10,-10,-10,-10,-5,-5,-5,-5,
                             -5,0,0,0,0,0,5,5,5,5,5,10,10,10,
                              10,10};
std::vector <Lps::Real64> yCtlPts;
for (i = 0 ; i < 25 ; ++i)
  yCtlPts.push_back (yCtlPtArray[i]);
```

```
// create y SplineRangeVariable
Lps::SplineRangeVariable yEntry;
yEntry.create (yCtlPts, "y");

// create STL vector of z control points
Lps::Real64 zCtlPtArray[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0};
std::vector <Lps::Real64> zCtlPts;
for (i = 0 ; i < 25 ; ++i)
  zCtlPts.push_back (zCtlPtArray[i]);

// create z SplineRangeVariable
Lps::SplineRangeVariable zEntry;
zEntry.create (zCtlPts, "z");

// create STL vector of range variables
std::vector<Lps::SplineRangeVariable> rangeData;
rangeData.push_back (xEntry);
rangeData.push_back (yEntry);
rangeData.push_back (zEntry);

Lps::SurfacePtr surface;
surface = structure->createSurface ("bf41", 1, domainData,
                                    rangeData, weights);
```

Other methods to create a Surface are also available and can be found in the LEAPS reference manual [4].

Retrieving a Surface

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a Surface with the following code.

```
Lps::SurfacePtr surface;
surface = structure->getSurface ("bf41", 1);
```

Destroying a Surface

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a Surface with the following code.

```
structure->destroySurface ("bf41", 1);
```

Determining the Existence of a Surface

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a Surface exists with the following code.

```
If (structure->doesSurfaceExist ("bf41", 1)
  std::cout << "Surface Exists" << std::endl;
else
  std::cout << "Surface does NOT Exist" << std::endl;
```

Listing the Surfaces Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the Surfaces unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfSurfaces ();
```

or a list of names and versions of the Surfaces with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfSurfaces ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a Pcurve

A LEAPS Pcurve is parametric curve that lies on a LEAPS Surface. It is defined by a non-uniform rational B-spline (NURBS). The spline is defined by a STL vector of SplineDomainVaribles and a STL vector of SplineRangeVariables. If the spline is rational a STL vector of weights is also part of the spline. A SplineDomainVariable is defined with a label, the order, and the knots associated with the domain variable. A SplineRangeVariable is defined by a label and the control points associated with the range variable. There one SplineDomainVariable that is s parametric variable of the spline and two SplineRangeVariables that are the u and v control points. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a Pcurve with the following code.

```
// define the order of the domain variable
Lps::Uint32 order4 = 4;

// define array of knots
Lps::Real64 knotArray[9] = {0, 0, 0, 0, 0.5, 1, 1, 1, 1};
std::vector <Lps::Real64> knots;
for (int i = 0 ; i < 9 ; ++i)
  knots.push_back (knotArray[i]);

// create s domain variable
Lps::SplineDomainVariable sEntry;
sEntry.create (knots, order4, "s");

// create the STL vector of domain variables
std::vector<Lps::SplineDomainVariable> domainData;
domainData.push_back (sEntry);

// create STL vector of weights
std::vector <Lps::Real64> weights;
for (i = 0 ; i < 5 ; ++i)
  weights.push_back (1.0);

// create STL vector of u control points
Lps::Real64 uCtlPtArray[] = {0,.25,.5,.75,1};
std::vector <Lps::Real64> uCtlPts;
for (i = 0 ; i < 5 ; ++i)
  uCtlPts.push_back (uCtlPtArray[i]);

// create u SplineRangeVariable
Lps::SplineRangeVariable uEntry;
uEntry.create (uCtlPts, "u");
```

```
// create STL vector of v control points
Lps::Real64 vCtlPtArray[] = {0,0,0,0,0};
std::vector <Lps::Real64> vCtlPts;
for (i = 0 ; i < 5 ; ++i)
  vCtlPts.push_back (vCtlPtArray[i]);

// create v SplineRangeVariable
Lps::SplineRangeVariable vEntry;
vEntry.create (vCtlPts, "v");

// create STL vector of range variables
std::vector<Lps::SplineRangeVariable> rangeData;
rangeData.push_back (uEntry);
rangeData.push_back (vEntry);

// retrieve surface that the curve lies on
Lps::SurfacePtr mappedTo;
mappedTo = structure->getSurface ("bf41", 1);

Lps::PcurvePtr Pcurve;
Pcurve = structure->createPcurve ("bf1", 1, mappedTo, domainData,
                                  rangeData, weights);
```

Other methods to create a Pcurve are also available and can be found in the LEAPS reference manual [4].

Retrieving a Pcurve

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a Pcurve with the following code.

```
Lps::PcurvePtr Pcurve;
Pcurve = structure->getPcurve ("bf1", 1);
```

Destroying a Pcurve

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a Pcurve with the following code.

```
structure->destroyPcurve ("bf1", 1);
```

Determining the Existence of a Pcurve

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a Pcurve exists with the following code.

```
If (structure->doesPcurveExist ("bf1", 1)
  std::cout << "Pcurve Exists" << std::endl;
else
  std::cout << "Pcurve does NOT Exist" << std::endl;
```

Listing the Pcurves Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the Pcurves unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfPcurves ();
```

or a list of names and versions of the Pcurves with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfPcurves ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a Ppoint

A LEAPS Ppoint is parametric point that lies on a LEAPS Pcurve. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a Ppoint with the following code.

```
Lps::Real64 value = 0.0;
Lps::PcurvePtr mappedTo = structure->getPcurve ("bf1", 1);
Lps::PpointPtr ppoint;
ppoint = structure->createPpoint ("ep1", 1, mappedTo, value);
```

Other methods to create a Ppoint are also available and can be found in the LEAPS reference manual [4].

Retrieving a Ppoint

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a Ppoint with the following code.

```
Lps::PpointPtr ppoint;
ppoint = structure->getPpoint ("ep1", 1);
```

Destroying a Ppoint

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a Ppoint with the following code.

```
structure->destroyPpoint ("ep1", 1);
```

Determining the Existence of a Ppoint

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a Ppoint exists with the following code.

```
If (structure->doesPpointExist ("ep1", 1)
  std::cout << "Ppoint Exists" << std::endl;
else
  std::cout << "Ppoint does NOT Exist" << std::endl;
```

Listing the Ppoints Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the Ppoints unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfPpoints ();
```

or a list of names and versions of the Ppoints with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfPpoints ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a CoPoint

A LEAPS CoPoint is a group of LEAPS Ppoint objects that are associated with the same location in model space. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a CoPoint with the following code.

```
// create the cartesian location at which the Ppoints located
Lps::CartesianLocation pt (0.0, -10.0, 0.0);
// create a STL vector of the associated Ppoints
Lps::PpointPtrList associatedPpoints;
associatedPpoints.push_back (structure->getPpoint ("ep1_1", 1));
associatedPpoints.push_back (structure->getPpoint ("ep4_2", 1));
associatedPpoints.push_back (structure->getPpoint ("ep13_1", 1));
associatedPpoints.push_back (structure->getPpoint ("ep16_2", 1));
associatedPpoints.push_back (structure->getPpoint ("ep23_2", 1));
associatedPpoints.push_back (structure->getPpoint ("ep24_1", 1));

// create the CoPpoint
Lps::CoPointPtr coPoint;
coPoint = structure->createCoPoint ("cp1", 1, associatedPpoints,
                                    pt);
```

Other methods that create a CoPoint are also available and can be found in the LEAPS reference manual [4].

Retrieving a CoPoint

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a CoPoint with the following code.

```
Lps::CoPointPtr coPoint;
coPoint = structure->getCoPoint ("cp1", 1);
```

Destroying a CoPoint

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a CoPoint with the following code.

```
structure->destroyCoPoint ("cp1", 1);
```

Determining the Existence of a CoPoint

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a CoPoint exists with the following code.

```
If (structure->doesCoPointExist ("cp1", 1)
  std::cout << "CoPoint Exists" << std::endl;
else
  std::cout << "CoPoint does NOT Exist" << std::endl;
```

Listing the CoPoints Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the CoPoints unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfCoPoints ();
```

or a list of names and versions of the CoPoints with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfCoPoints ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating an Edge

A LEAPS Edge is an oriented segment of a LEAPS Pcurve object. The Edge is defined by specifying the start and end Ppoints of the Edge. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create an Edge with the following code.

```
Lps::PpointPtr startPt;
startPt = structure->getPpoint ("ep1_1", 1);

Lps::PpointPtr endPt;
endPt = structure->getPpoint ("ep1_2", 1);

// create the Edge
Lps::EdgePtr edge;
edge = structure->createEdge ("ec1_1a", 1, startPt, endPt);
```

Other methods that create an Edge are also available and can be found in the LEAPS reference manual [4].

Retrieving an Edge

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve an Edge with the following code.

```
Lps::EdgePtr edge;
edge = structure->getEdge ("ec1_1a", 1);
```

Destroying an Edge

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy an Edge with the following code.

```
structure->destroyEdge ("ec1_1a", 1);
```

Determining the Existence of an Edge

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if an Edge exists with the following code.

```
If (structure->doesEdgeExist ("ec1_1a", 1)
  std::cout << "Edge Exists" << std::endl;
else
  std::cout << "Edge does NOT Exist" << std::endl;
```

Listing the Edges Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the Edges unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfEdges ();
```

or a list of names and versions of the Edges with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfEdges ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a CoEdge

A LEAPS CoEdge is a group of LEAPS Edge objects that are associated with the same location in model space. A CoEdge generally repesents the intersection of two surfaces. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a CoEdge with the following code.

```
// create a STL vector of the associated Edges
Lps::EdgePtrList associatedEdges;
associatedEdges.push_back (structure->getEdge ("ec1_1a", 1));
associatedEdges.push_back (structure->getEdge ("ec13_1a", 1));

// create the CoEdge
Lps::CoEdgePtr coEdge;
coEdge = structure->createCoEdge ("ce1", 1, associatedEdges)
```

Other methods that create a CoEdge are also available and can be found in the LEAPS reference manual [4].

Retrieving a CoEdge

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a CoEdge with the following code.

```
Lps::CoEdgePtr coEdge;
coEdge = structure->getCoEdge ("ce1", 1);
```

Destroying a CoEdge

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a CoEdge with the following code.

```
structure->destroyCoEdge ("ce1", 1);
```

Determining the Existence of a CoEdge

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a CoEdge exists with the following code.

```
If (structure->doesCoEdgeExist ("ce1", 1)
  std::cout << "CoEdge Exists" << std::endl;
else
  std::cout << "CoEdge does NOT Exist" << std::endl;
```

Listing the CoEdges Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the CoEdges unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfCoEdges ();
```

or a list of names and versions of the CoEdges with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfCoEdges ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating an EdgeLoop

A LEAPS EdgeLoop is a set of connected LEAPS Edge objects that form a closed loop that is not self intersecting. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create an EdgeLoop with the following code.

```
// create a STL vector of the connected Edges
Lps::EdgePtrList connectedEdges;
connectedEdges.push_back (structure->getEdge ("ec1_1a", 1));
connectedEdges.push_back (structure->getEdge ("ec5_2a", 1));
connectedEdges.push_back (structure->getEdge ("ec5_1a", 1));
connectedEdges.push_back (structure->getEdge ("ec3_2a", 1));
connectedEdges.push_back (structure->getEdge ("ec4_1a", 1));
```

```
// create the EdgeLoop
Lps::EdgeLoopPtr edgeLoop;
edgeLoop = structure->createEdgeLoop ("el_1a", 1, connectedEdges)
```

Other methods that create an EdgeLoop are also available and can be found in the LEAPS reference manual [4].

Retrieving an EdgeLoop

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve an EdgeLoop with the following code.

```
Lps::EdgeLoopPtr edgeLoop;
edgeLoop = structure->getEdgeLoop ("el_1a", 1);
```

Destroying an EdgeLoop

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy an EdgeLoop with the following code.

```
structure->destroyEdgeLoop ("el_1a", 1);
```

Determining the Existence of an EdgeLoop

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if an EdgeLoop exists with the following code.

```
If (structure->doesEdgeLoopExist ("el_1a", 1)
  std::cout << "EdgeLoop Exists" << std::endl;
else
  std::cout << "EdgeLoop does NOT Exist" << std::endl;
```

Listing the EdgeLoops Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the EdgeLoops unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfEdgeLoops ();
```

or a list of names and versions of the EdgeLoops with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfEdgeLoops ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a Face

A LEAPS Face is a trimmed NURBS surface. It is defined by a LEAPS EdgeLoop that is the outer boundary of the Face and any number of non-intersecting EdgeLoops that are holes in the Face. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create an Face with the following code.

```
// retrieve the EdgeLoop that is the outer boundary of the Face
Lps::EdgeLoopPtr outerLoop;
outerLoop = structure->getEdgeLoop ("el_1a", 1);

// no holes – create empty STL vector of EdgeLoops
Lps::EdgeLoopPtrList innerLoops;

// create the Face
Lps::FacePtr face;
face = structure->createFace ("fa1", 1, outerLoop, innerLoops);
```

Other methods that create a Face are also available and can be found in the LEAPS reference manual [4].

Retrieving a Face

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a Face with the following code.

```
Lps::FacePtr face;
face = structure->getFace ("fa1", 1);
```

Destroying a Face

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a Face with the following code.

```
structure->destroyFace ("fa1", 1);
```

Determining the Existence of a Face

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a Face exists with the following code.

```
If (structure->doesFaceExist ("fa1", 1)
  std::cout << "Face Exists" << std::endl;
else
  std::cout << "Face does NOT Exist" << std::endl;
```

Listing the Faces Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the Faces unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfFaces ();
```

or a list of names and versions of the Faces with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfFaces ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating an OrientedClosedShell

A LEAPS OrientedClosedShell is a set of LEAPS Face objects that form a closed shell that is oriented. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create an OrientedClosedShell with the following code.

```
// create a STL vector of the connected Faces
Lps::FacePtrList connectedFaces;
connectedFaces.push_back (structure->getFace ("fa1", 1));
connectedFaces.push_back (structure->getFace ("fa23", 1));
connectedFaces.push_back (structure->getFace ("fa29", 1));
connectedFaces.push_back (structure->getFace ("fa17", 1));
connectedFaces.push_back (structure->getFace ("fa31", 1));
connectedFaces.push_back (structure->getFace ("fa9", 1));

// create the OrientedClosedShell
Lps::OrientedClosedShellPtr shell;
shell = structure->createOrientedClosedShell ("os_comp1", 1,
                                              connectedFaces)
```

Other methods that create an OrientedClosedShell are also available and can be found in the LEAPS reference manual [4].

Retrieving an OrientedClosedShell

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve an OrientedClosedShell with the following code.

```
Lps::OrientedClosedShellPtr shell;
shell = structure->getOrientedClosedShell ("os_comp1", 1);
```

Destroying an OrientedClosedShell

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy an OrientedClosedShell with the following code.

```
structure->destroyOrientedClosedShell ("os_comp1", 1);
```

Determining the Existence of an OrientedClosedShell

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if an OrientedClosedShell exists with the following code.

```
If (structure->doesOrientedClosedShellExist ("os_comp1", 1)
  std::cout << "OrientedClosedShell Exists" << std::endl;
```

```
else
  std::cout << "OrientedClosedShell does NOT Exist" << std::endl;
```

Listing the OrientedClosedShells Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the OrientedClosedShells unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList;
uidList = structure->getUidsOfOrientedClosedShells ();
```

or a list of names and versions of the OrientedClosedShells with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfOrientedClosedShells ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a Solid

A LEAPS Solid is a boundary-represented (BREP) solid. It is defined by a LEAPS OrientedClosedShell object that is the outer boundary of the Solid and any number of non-intersecting OrientedClosedShell objects that are voids in the Solid. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a Solid with the following code.

```
// retrieve the OrientedClosedShell that is the outer boundary
// of the Solid
Lps::OrientedClosedShellPtr outerShell;
outerShell = structure->getOrientedClosedShell ("os_comp1", 1);

// no voids – create empty STL vector of OrientedClosedShells
Lps::OrientedClosedShellPtrList voidShells;

// create the Solid
Lps::SolidPtr solid;
solid = structure->createSolid ("so_comp1", 1, outerShell,
                                voidShells);
```

Other methods that create a Solid are also available and can be found in the LEAPS reference manual [4].

Retrieving a Solid

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a Solid with the following code.

```
Lps::SolidPtr solid;
solid = structure->getSolid ("so_comp1", 1);
```

Destroying a Solid

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a Solid with the following code.

```
structure->destroySolid ("so_comp1", 1);
```

Determining the Existence of a Solid

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a Solid exists with the following code.

```
If (structure->doesSolidExist ("so_comp1", 1)
  std::cout << "Solid Exists" << std::endl;
else
  std::cout << "Solid does NOT Exist" << std::endl;
```

Listing the Solids Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the Solids unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfSolids ();
```

or a list of names and versions of the Solids with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfSolids ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a TopologicalView

A LEAPS TopologicalView is a view of the geometry that typically represents traditional CAD entities. In particular, a TopologicalView is a LEAPS Solid, Face, or Surface that has properties. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a TopologicalView that is a Face with the following code.

```
// retrieve the Face that the TopologicalView represents
Lps::FacePtr face;
face = structure->getFace ("fa17", 1);

// create the TopologicalView
Lps::TopologicalViewPtr tView;
tView = structure->createTopologicalView ("topv_hulls_plate1", 1,
                                          face);
```

Other methods that create a TopologicalView are also available and can be found in the LEAPS reference manual [4].

Retrieving a TopologicalView

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a TopologicalView with the following code.

```
Lps::TopologicalViewPtr tView;
tView = structure->getTopologicalView ("topv_hulls_plate1", 1);
```

Destroying a TopologicalView

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a TopologicalView with the following code.

```
structure->destroyTopologicalView ("topv_hulls_plate1", 1);
```

Determining the Existence of a TopologicalView

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a TopologicalView exists with the following code.

```
If (structure->doesTopologicalViewExist ("topv_hulls_plate1", 1)
  std::cout << "TopologicalView Exists" << std::endl;
else
  std::cout << "TopologicalView does NOT Exist" << std::endl;
```

Listing the TopologicalViews Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the TopologicalViews unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList;
uidList = structure->getUidsOfTopologicalViews ();
```

or a list of names and versions of the TopologicalViews with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfTopologicalViews ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a CommonView

A LEAPS CommonView is group of LEAPS TopologicalView objects and LEAPS CommonView objects that represent a logical view of the geometry. If structure is a Structure object that has been retrieved from a Concept or a Component, a program can create a CommonView with the following code.

```
// retrieve TopologicalViews that compose the CommonView
Lps::TopologicalViewPtrList topViews;
```

```
topViews.push_back (structure->getTopologicalView
                                        ("topv_deck1_plate1", 1));
topViews.push_back (structure->getTopologicalView
                                        ("topv_deck2_plate1", 1));
topViews.push_back (structure->getTopologicalView
                                        ("topv_hullp_plate1", 1));
topViews.push_back (structure->getTopologicalView
                                        ("topv_hulls_plate1", 1));
topViews.push_back (structure->getTopologicalView
                                        ("topv_trans2_plate1", 1));
topViews.push_back (structure->getTopologicalView
                                        ("topv_trans2_plate2", 1));

// no CommonViews that are a part of this CommonView – create
// empty STL vector of CommonViews
 Lps::CommonViewsPtrList comViews;

// create the CommonView
Lps::CommonViewPtr commonView;
commonView = structure->createCommonView ("comv_comp1", 1,
                                          topViews, comViews);
```

Other methods that create a CommonView are also available and can be found in the LEAPS reference manual [4].

Retrieving a CommonView

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can retrieve a CommonView with the following code.

```
Lps::CommonViewPtr commonView;
commonView = structure->getCommonView ("comv_comp1", 1);
```

Destroying a CommonView

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can destroy a CommonView with the following code.

```
structure->destroyCommonView ("comv_comp1", 1);
```

Determining the Existence of a CommonView

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can determine if a CommonView exists with the following code.

```
If (structure->doesCommonViewExist ("comv_comp1", 1)
  std::cout << "CommonView Exists" << std::endl;
else
  std::cout << "CommonView does NOT Exist" << std::endl;
```

Listing the CommonViews Contained in a Structure

If structure is a Structure object that has been retrieved from a Concept or a Component, a program can obtain a list of the CommonViews unique identifiers that are contained within the Structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfCommonViews ();
```

or a list of names and versions of the CommonViews with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfCommonViews ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing Materials for LEAPS Objects

Materials are represented by LEAPS Material objects and LEAPS MaterialGroup objects. These objects are associated with the the LEAPS Structure class, the LEAPS CommonView class, and the LEAPS TopologicalView class. These classes manage LEAPS Material and MaterialGroup objects that are associated with them. These classes have member methods that provide various functions that involve Material and MaterialGroup objects. These member methods can:

- Create a Material or MaterialGroup object,
- Retrieve a Material or MaterialGroup object,
- Destroy a Material or MaterialGroup object,
- Determine the existence of a Material or MaterialGroup object, and
- List Material and MaterialGroup objects that are managed by the class.

Creating a Material for a LEAPS Object

If the LEAPS object has materials, a method with the name "createMaterial" is available to create a Material. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can create a Material to be associated with the structure with the following code.

```
// create material for the structure
Lps::MaterialPtr hySteel;
hySteel = structure->createMaterial ("hySteel", 1);
```

Retrieving a Material for a LEAPS Object

If the LEAPS object has materials, a method with the name "getMaterial" is available to retrieve a Material. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can retrieve a Material that has been associated with the structure with the following code.

```
Lps::MaterialPtr hySteel;
hySteel = structure->getMaterial ("hySteel", 1);
```

Destroying a Material for a LEAPS Object

If the LEAPS object has materials, a method with the name "destroyMaterial" is available to destroy a Material. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can destroy a Material that has been associated with the structure with the following code.

```
structure->destroyMaterial ("hySteel", 1);
```

Determining the Existence of a Material for a LEAPS Object

If the LEAPS object has materials, a method with the name "doesMaterialExists" is available to determine if a Material is associated with the LEAPS object. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can determine if a Material is associated with the structure with the following code.

```
If (structure->doesMaterialExist ("hySteel", 1)
  std::cout << "Material Exists" << std::endl;
else
  std::cout << "Material does NOT Exist" << std::endl;
```

Listing the Materials Managed by a LEAPS Object

If the LEAPS object has materials, a method with the name "getUidsOfMaterials" is available to retrieve a list of the unique identifiers of the Material objects that are associated with the LEAPS object. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can retrieve the list of unique identifiers of the Material objects associated with the structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfMaterials ();
```

or a list of names and versions of the Material objects with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfMaterials ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a MaterialGroup for a LEAPS Object

If the LEAPS object has materials, a method with the name "createMaterialGroup" is available to create a MaterialGroup. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can create a MaterialGroup to be associated with structure with the following code.

```
// create a material group for a composite for the structure
Lps::MaterialGroupPtr composite;
composite = structure->createMaterialGroup ("Composite", 1,);
```

Retrieving a MaterialGroup for a LEAPS Object

If the LEAPS object has materials, a method with the name "getMaterialGroup" is available to retrieve a MaterialGroup. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can retrieve a MaterialGroup that has been associated with the structure with the following code.

```
Lps::MaterialGroupPtr composite;
composite = structure->getMaterialGroup ("Composite", 1);
```

Destroying a MaterialGroup for a LEAPS Object

If the LEAPS object has materials, a method with the name "destroyMaterialGroup" is available to destroy a MaterialGroup. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can destroy a MaterialGroup that has been associated with the structure with the following code.

```
structure->destroyMaterialGroup ("Composite", 1);
```

Determining the Existence of a MaterialGroup for a LEAPS Object

If the LEAPS object has materials, a method with the name "doesMaterialGroupExists" is available to determine if a MaterialGroup is associated with the LEAPS object. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can determine if a MaterialGroup is associated with the structure with the following code.

```
If (structure->doesMaterialGroupExist ("Composite", 1)
  std::cout << "MaterialGroup Exists" << std::endl;
else
  std::cout << "MaterialGroup does NOT Exist" << std::endl;
```

Listing the MaterialGroups Managed by a LEAPS Object

If the LEAPS object has materials, a method with the name "getUidsOfMaterialGroups" is available to retrieve a list of the unique identifiers of the MaterialGroup objects that are associated with the LEAPS object. If structure is a LEAPS Structure object that has been retrieved from a Concept, a program can retrieve the list of unique identifiers of the MaterialGroup objects associated with the structure with the following code.

```
Lps::UniqueIdList uidList = structure->getUidsOfMaterialGroups ();
```

or a list of names and versions of the MaterialGroup objects with

```
Lps::NameVersionPairList nvList;
nvList = structure->getNameVersionPairsOfMaterialGroups ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Managing Properties for LEAPS Objects

Properties are represented by LEAPS Property objects and LEAPS PropertyGroup objects. These objects are associated with the primary LEAPS classes, the LEAPS Structure class, the LEAPS CommonView class, and the LEAPS TopologicalView class. These classes manage LEAPS Property and PropertyGroup objects that are associated with them. These classes have member methods that provide various functions that involve Property and PropertyGroup objects. These member methods can:

- Create a Property or PropertyGroup object,
- Retrieve a Property or PropertyGroup object,
- Destroy a Property or PropertyGroup object,
- Determine the existence of a Property or PropertyGroup object, and
- List Property and PropertyGroup objects that are managed by the class.

Creating a Property for a LEAPS Object

If the LEAPS object has properties, a method with the name "createProperty" is available to create a Property. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can create a Property to be associated with concept with the following code.

```
// create the PropertyData that contains design waterline
Lps::PropertyDataPtr dwl = new Lps::RealScalar (25.5);

// create design waterline property for the concept
Lps::PropertyPtr dwlProp;
dwlProp = concept->createProperty ("DWL", 1, dwl);
```

Retrieving a Property for a LEAPS Object

If the LEAPS object has properties, a method with the name "getProperty" is available to retrieve a Property. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can retrieve a Property that has been associated with the concept with the following code.

```
Lps::PropertyPtr dwlProp;
dwlProp = concept->getProperty ("DWL", 1);
```

Destroying a Property for a LEAPS Object

If the LEAPS object has properties, a method with the name "destroyProperty" is available to destroy a Property. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can destroy a Property that has been associated with the concept with the following code.

```
concept->destroyProperty ("DWL", 1);
```

Determining the Existence of a Property for a LEAPS Object

If the LEAPS object has properties, a method with the name "doesPropertyExists" is available to determine if a Property is associated with the LEAPS object. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can determine if a Property is associated with the concept with the following code.

```
If (concept->doesPropertyExist ("DWL", 1)
  std::cout << "Property Exists" << std::endl;
else
  std::cout << "Property does NOT Exist" << std::endl;
```

Listing the Properties Managed by a LEAPS Object

If the LEAPS object has properties, a method with the name "getUidsOfProperties" is available to retrieve a list of the unique identifiers of the Property objects that are associated with the LEAPS object. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can retrieve the list of unique identifiers of the Property objects associated with the concept with the following code.

```
Lps::UniqueIdList uidList = concept->getUidsOfProperties ();
```

or a list of names and versions of the Property objects with

```
Lps::NameVersionPairList nvList;
nvList = concept->getNameVersionPairsOfProperties ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

Creating a PropertyGroup for a LEAPS Object

If the LEAPS object has properties, a method with the name "createPropertyGroup" is available to create a PropertyGroup. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can create a PropertyGroup to be associated with concept with the following code.

```
// retrieve properties needed to create property group
Lps::PropertyPtrList propList;
propList.push_back (concept->getPropertyGroup ("aftPerpAtRdr", 1);
propList.push_back (concept->getPropertyGroup ("aftPerpAtDwl", 1);

// no PropertyGroups that are a part of this PropertyGroup –
// create empty STL vector of PropertyGroups
Lps::PropertyGroupPtrList propGroupList;

// create a property group for aft perpendicular view for
// the concept
Lps::PropertyGroupPtr aftPerp;
aftPerp = concept->createPropertyGroup ("AftPerp", 1,
                                        propList, propGroupList);
```

Retrieving a PropertyGroup for a LEAPS Object

If the LEAPS object has properties, a method with the name "getPropertyGroup" is available to retrieve a PropertyGroup. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can retrieve a PropertyGroup that has been associated with the concept with the following code.

```
Lps::PropertyGroupPtr aftPerp;
aftPerp = concept->getPropertyGroup ("AftPerp", 1);
```

Destroying a PropertyGroup for a LEAPS Object

If the LEAPS object has properties, a method with the name "destroyPropertyGroup" is available to destroy a PropertyGroup. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can destroy a PropertyGroup that has been associated with the concept with the following code.

```
concept->destroyPropertyGroup ("AftPerp", 1);
```

Determining the Existence of a PropertyGroup for a LEAPS Object

If the LEAPS object has properties, a method with the name "doesPropertyGroupExists" is available to determine if a PropertyGroup is associated with the LEAPS object. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can determine if a PropertyGroup is associated with the concept with the following code.

```
If (concept->doesPropertyGroupExist ("AftPerp", 1)
  std::cout << "PropertyGroup Exists" << std::endl;
else
  std::cout << "PropertyGroup does NOT Exist" << std::endl;
```

Listing the PropertyGroups Managed by a LEAPS Object

If the LEAPS object has properties, a method with the name "getUidsOfPropertyGroups" is available to retrieve a list of the unique identifiers of the PropertyGroup objects that are associated with the LEAPS object. If concept is a LEAPS Concept object that has been retrieved from a Study, a program can retrieve the list of unique identifiers of the PropertyGroup objects associated with the concept with the following code.

```
Lps::UniqueIdList uidList = concept->getUidsOfPropertyGroups ();
```

or a list of names and versions of the PropertyGroup objects with

```
Lps::NameVersionPairList nvList;
nvList = concept->getNameVersionPairsOfPropertyGroups ();
```

UniqueIdList is a type definition for a Standard Template Library (**STL**) vector of strings. NameVersionPairList is a type definition for a STL vector of pairs where a pair consists of a string and an unsigned integer.

DETERMINING THE CONTENTS OF LEAPS OBJECTS

Determining the Contents of a LEAPS Database

A LEAPS Factory is created to manage a LEAPS database. At the highest level, a LEAPS database is composed of Study objects and Catalog objects. Currently methods associated with Catalog objects have not been implemented. If 'leapsDB' is a Factory created to manage a LEAPS database, the following code returns the number of Study objects contained in the database, lists the unique identifiers of the objects, and then retrieves them individually.

```
// find how many studies are in the LEAPS database
Lps::Uint32 studyCount;
studyCount = leapsDB->numberOfStudies ();
std::cout << "Number of Studies: " << studyCount << std::endl;

// get unique identifiers of the studies in the LEAPS database and
// print them
Lps::UniqueIdList uidList = leapsDB->getUidsOfStudies ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Study " << *it << std::endl;

// given the list of unique identifiers, retrieve studies
// individually and store in a STL vector of StudyPtr
Lps::StudyPtrList studies;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  studies.push_back (leapsDB->getStudy (*it));
```

Determining the Contents of a LEAPS Study Object

A LEAPS Study is composed of Concept objects, Scenario objects, Property objects, and PropertyGroup objects. If 'cvxStudy' is a Study that has been retrieved from a database, the contents of the Study object can be queried and retrieved.

Determining the Concepts of a Study Object

The following code returns the number of Concept objects contained by the Study object 'cvxStudy'', lists the unique identifiers of the Concept objects, and then retrieves them individually.

```
// find how many concepts are in the Study object
Lps::Uint32 conceptCount;
conceptCount = cvxStudy->numberOfConcepts ();
std::cout << "Number of Concepts: " << conceptCount << std::endl;

// get unique identifiers of the concepts in the Study and
// print them
Lps::UniqueIdList uidList = cvxStudy->getUidsOfConcepts ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Concept " << *it << std::endl;
```

```
// given the list of unique identifiers, retrieve concepts
// individually and store in a STL vector of ConceptPtr
Lps::ConceptPtrList concepts;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  concepts.push_back (cvxStudy->getConcept (*it));
```

Determining the Scenarios of a Study Object

The following code returns the number of Scenario objects contained by the Study object 'cvxStudy'', lists the unique identifiers of the Scenario objects, and then retrieves them individually.

```
// find how many scenarios are in the Study object
Lps::Uint32 scenarioCount;
scenarioCount = cvxStudy->numberOfScenarios ();
std::cout << "Number of Scenarios: " << scenarioCount
          << std::endl;

// get unique identifiers of the scenarios in the Study and
// print them
Lps::UniqueIdList uidList = cvxStudy->getUidsOfScenarios ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Scenario " << *it << std::endl;

// given the list of unique identifiers, retrieve scenarios
// individually and store in a STL vector of ScenarioPtr
Lps::ScenarioPtrList scenarios;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  scenarios.push_back (cvxStudy->getScenario (*it));
```

Determining the Properties of a Study Object

The following code returns the number of Property objects contained by the Study object 'cvxStudy'', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many propertys are in the Study object
Lps::Uint32 propertyCount;
propertyCount = cvxStudy->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the Study and
// print them
Lps::UniqueIdList uidList = cvxStudy->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (cvxStudy->getProperty (*it));
```

Determining the PropertyGroups of a Study Object

The following code returns the number of PropertyGroup objects contained by the Study object 'cvxStudy'', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the Study object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = cvxStudy->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the Study and
// print them
Lps::UniqueIdList uidList = cvxStudy->getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (cvxStudy->getPropertyGroup (*it));
```

Determining the Contents of a LEAPS Concept Object

A LEAPS Concept is composed of Component objects, System objects, Property objects, PropertyGroup objects, and a concept Structure object. If 'cvx' is a Concept that has been retrieved from a Study, the contents of the Concept object can be queried and retrieved.

Determining the Components of a Concept Object

The following code returns the number of Component objects contained by the Concept object 'cvx'', lists the unique identifiers of the Component objects, and then retrieves them individually.

```
// find how many components are in the Concept object
Lps::Uint32 componentCount;
componentCount = cvx->numberOfComponents ();
std::cout << "Number of Components: " << componentCount
          << std::endl;

// get unique identifiers of the components in the Concept and
// print them
Lps::UniqueIdList uidList = cvx->getUidsOfComponents ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Concept " << *it << std::endl;

// given the list of unique identifiers, retrieve components
// individually and store in a STL vector of ComponentPtr
Lps::ComponentPtrList components;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  components.push_back (cvx->getComponent (*it));
```

Determining the Systems of a Concept Object

The following code returns the number of System objects contained by the Concept object 'cvx'', lists the unique identifiers of the System objects, and then retrieves them individually.

```
// find how many systems are in the Concept object
Lps::Uint32 systemCount;
systemCount = cvx->numberOfSystems ();
std::cout << "Number of Systems: " << systemCount
          << std::endl;

// get unique identifiers of the systems in the Concept and
// print them
Lps::UniqueIdList uidList = cvx->getUidsOfSystems ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Concept " << *it << std::endl;

// given the list of unique identifiers, retrieve systems
// individually and store in a STL vector of SystemPtr
Lps::SystemPtrList systems;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  systems.push_back (cvx->getSystem (*it));
```

Determining the Properties of a Concept Object

The following code returns the number of Property objects contained by the Concept object 'cvx', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the Concept object
Lps::Uint32 propertyCount;
propertyCount = cvx->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the Concept and
// print them
Lps::UniqueIdList uidList = cvx->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (cvx->getProperty (*it));
```

Determining the PropertyGroups of a Concept Object

The following code returns the number of PropertyGroup objects contained by the Concept object 'cvx', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the Concept object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = cvx->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
```

```
          << std::endl;

// get unique identifiers of the propertyGroups in the Concept and
// print them
Lps::UniqueIdList uidList = cvx->getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (cvx->getPropertyGroup (*it));
```

Retrieving a Concept's Structure

The following code retrieves the concept's structure from the Concept object 'cvx' that has been retrieved from a Study.

```
Lps::StructurePtr cvxStructure;
cvxStructure = cvx->getConceptStructure ();
```

Determining the Contents of a LEAPS Component Object

A LEAPS Component is composed of Property objects, PropertyGroup objects, and a component Structure object. If 'pump' is a Component that has been retrieved from a Concept, the contents of the Component object can be queried and retrieved.

Determining the Properties of a Component Object

The following code returns the number of Property objects contained by the Component object 'pump', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the Component object
Lps::Uint32 propertyCount;
propertyCount = pump->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the Component and
// print them
Lps::UniqueIdList uidList = pump->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (pump->getProperty (*it));
```

Determining the PropertyGroups of a Component Object

The following code returns the number of PropertyGroup objects contained by the Component object 'pump', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the Component object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = pump->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// Component and print them
Lps::UniqueIdList uidList = pump->getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (pump->getPropertyGroup (*it));
```

Retrieving a Component's Structure

The following code retrieves the component's structure from the Component object 'pump' that has been retrieved from a Concept.

```
Lps::StructurePtr pumpStructure;
pumpStructure = pump->getComponentStructure ();
```

Determining the Contents of a LEAPS System Object

A LEAPS System is composed of Property objects, PropertyGroup objects, Diagram objects, and an aggregate Component object. The aggregate component is a Component object that represents the System as a component. Currently, the methods to access Diagram objects and the aggregate Component object is not implemented. If 'fireMain' is a System that has been retrieved from a Concept, the contents of the System object can be queried and retrieved.

Determining the Properties of a System Object

The following code returns the number of Property objects contained by the System object 'fireMain', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the System object
Lps::Uint32 propertyCount;
propertyCount = fireMain->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the System and
// print them
Lps::UniqueIdList uidList = fireMain->getUidsOfProperties ();
```

```
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (fireMain->getProperty (*it));
```

Determining the PropertyGroups of a System Object

The following code returns the number of PropertyGroup objects contained by the System object 'fireMain', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the System object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = fireMain->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// System and print them
Lps::UniqueIdList uidList = fireMain->getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (fireMain->getPropertyGroup (*it));
```

Determining the Contents of a LEAPS Scenario Object

A LEAPS Scenario is currently composed of Property objects and PropertyGroup objects. The LEAPS Scenario class will be expanded in the future. If 'planA' is a Scenario that has been retrieved from a Study, the contents of the Scenario object can be queried and retrieved.

Determining the Properties of a Scenario Object

The following code returns the number of Property objects contained by the Scenario object 'planA', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the Scenario object
Lps::Uint32 propertyCount;
propertyCount = planA->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the Scenario and
// print them
Lps::UniqueIdList uidList = planA->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;
```

```
// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (planA->getProperty (*it));
```

Determining the PropertyGroups of a Scenario Object

The following code returns the number of PropertyGroup objects contained by the Scenario object 'planA', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the Scenario object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = planA->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// Scenario and print them
Lps::UniqueIdList uidList = planA->getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (planA->getPropertyGroup (*it));
```

Determining the Contents of a LEAPS Structure Object

A LEAPS Structure is composed or has the following LEAPS objects:

- Property objects,
- PropertyGroup objects,
- Material objects,
- MaterialGroup objects,
- CommonView objects,
- TopologicalView objects,
- Solid objects,
- OrientedClosedShell objects,
- Face objects,
- EdgeLoop objects,
- CoEdge objects,
- Edge objects,
- CoPoint objects,
- Ppoint objects,
- Pcurve objects, and
- Surface objects.

A Structure object represents the geometry and the views of the geometry of either a Concept or a Component. If 'geom' is a Structure object that

was retrieved from a Concept or Component object, the contents of the Structure object can be queried and retrieved.

Determining the Properties of a Structure Object

The following code returns the number of Property objects contained by the Structure object 'geom', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the Structure object
Lps::Uint32 propertyCount;
propertyCount = geom->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the Structure and
// print them
Lps::UniqueIdList uidList = geom->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (geom->getProperty (*it));
```

Determining the PropertyGroups of a Structure Object

The following code returns the number of PropertyGroup objects contained by the Structure object 'geom', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the Structure object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = geom->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (geom->getPropertyGroup (*it));
```

Determining the Materials of a Structure Object

The following code returns the number of Material objects contained by the Structure object 'geom', lists the unique identifiers of the Material objects, and then retrieves them individually.

```
// find how many materials are in the Structure object
```

```
Lps::Uint32 materialCount;
materialCount = geom->numberOfMaterials ();
std::cout << "Number of Materials: " << materialCount
          << std::endl;

// get unique identifiers of the materials in the Structure and
// print them
Lps::UniqueIdList uidList = geom->getUidsOfMaterials ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Material " << *it << std::endl;

// given the list of unique identifiers, retrieve materials
// individually and store in a STL vector of MaterialPtr
Lps::MaterialPtrList materials;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  materials.push_back (geom->getMaterial (*it));
```

Determining the MaterialGroups of a Structure Object

The following code returns the number of MaterialGroup objects contained by the Structure object 'geom', lists the unique identifiers of the MaterialGroup objects, and then retrieves them individually.

```
// find how many materialGroups are in the Structure object
Lps::Uint32 materialGroupCount;
materialGroupCount = geom->numberOfMaterialGroups ();
std::cout << "Number of MaterialGroups: " << materialGroupCount
          << std::endl;

// get unique identifiers of the materialGroups in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfMaterialGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "MaterialGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve materialGroups
// individually and store in a STL vector of MaterialGroupPtr
Lps::MaterialGroupPtrList materialGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  materialGroups.push_back (geom->getMaterialGroup (*it));
```

Determining the CommonViews of a Structure Object

The following code returns the number of CommonView objects contained by the Structure object 'geom', lists the unique identifiers of the CommonView objects, and then retrieves them individually.

```
// find how many commonViews are in the Structure object
Lps::Uint32 commonViewCount;
commonViewCount = geom->numberOfCommonViews ();
std::cout << "Number of CommonViews: " << commonViewCount
          << std::endl;

// get unique identifiers of the commonViews in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfCommonViews ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "CommonView " << *it << std::endl;
```

```
// given the list of unique identifiers, retrieve commonViews
// individually and store in a STL vector of CommonViewPtr
Lps::CommonViewPtrList commonViews;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  commonViews.push_back (geom->getCommonView (*it));
```

Determining the TopologicalViews of a Structure Object

The following code returns the number of TopologicalView objects contained by the Structure object 'geom', lists the unique identifiers of the TopologicalView objects, and then retrieves them individually.

```
// find how many topologicalViews are in the Structure object
Lps::Uint32 topologicalViewCount;
topologicalViewCount = geom->numberOfTopologicalViews ();
std::cout << "Number of TopologicalViews: "
          << topologicalViewCount << std::endl;

// get unique identifiers of the topologicalViews in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfTopologicalViews ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "TopologicalView " << *it << std::endl;

// given the list of unique identifiers, retrieve topologicalViews
// individually and store in a STL vector of TopologicalViewPtr
Lps::TopologicalViewPtrList topologicalViews;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  topologicalViews.push_back (geom->getTopologicalView (*it));
```

Determining the Solids of a Structure Object

The following code returns the number of Solid objects contained by the Structure object 'geom', lists the unique identifiers of the Solid objects, and then retrieves them individually.

```
// find how many solids are in the Structure object
Lps::Uint32 solidCount;
solidCount = geom->numberOfSolids ();
std::cout << "Number of Solids: " << solidCount
          << std::endl;

// get unique identifiers of the solids in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfSolids ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Solid " << *it << std::endl;

// given the list of unique identifiers, retrieve solids
// individually and store in a STL vector of SolidPtr
Lps::SolidPtrList solids;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  solids.push_back (geom->getSolid (*it));
```

Determining the OrientedClosedShells of a Structure Object

The following code returns the number of OrientedClosedShell objects contained by the Structure object 'geom', lists the unique identifiers of the OrientedClosedShell objects, and then retrieves them individually.

```
// find how many orientedClosedShells are in the Structure object
Lps::Uint32 orientedClosedShellCount;
orientedClosedShellCount = geom->numberOfOrientedClosedShells ();
std::cout << "Number of OrientedClosedShells: "
          << orientedClosedShellCount << std::endl;

// get unique identifiers of the orientedClosedShells in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfOrientedClosedShells
();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "OrientedClosedShell " << *it << std::endl;

// given the list of unique identifiers, retrieve
orientedClosedShells
// individually and store in a STL vector of
OrientedClosedShellPtr
Lps::OrientedClosedShellPtrList orientedClosedShells;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  orientedClosedShells.push_back
                          (geom->getOrientedClosedShell (*it));
}
```

Determining the Faces of a Structure Object

The following code returns the number of Face objects contained by the Structure object 'geom', lists the unique identifiers of the Face objects, and then retrieves them individually.

```
// find how many faces are in the Structure object
Lps::Uint32 faceCount;
faceCount = geom->numberOfFaces ();
std::cout << "Number of Faces: " << faceCount
          << std::endl;

// get unique identifiers of the faces in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfFaces ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Face " << *it << std::endl;

// given the list of unique identifiers, retrieve faces
// individually and store in a STL vector of FacePtr
Lps::FacePtrList faces;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  faces.push_back (geom->getFace (*it));
```

Determining the EdgeLoops of a Structure Object

The following code returns the number of EdgeLoop objects contained by the Structure object 'geom', lists the unique identifiers of the EdgeLoop objects, and then retrieves them individually.

```
// find how many edgeLoops are in the Structure object
Lps::Uint32 edgeLoopCount;
edgeLoopCount = geom->numberOfEdgeLoops ();
std::cout << "Number of EdgeLoops: " << edgeLoopCount
          << std::endl;

// get unique identifiers of the edgeLoops in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfEdgeLoops ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "EdgeLoop " << *it << std::endl;

// given the list of unique identifiers, retrieve edgeLoops
// individually and store in a STL vector of EdgeLoopPtr
Lps::EdgeLoopPtrList edgeLoops;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  edgeLoops.push_back (geom->getEdgeLoop (*it));
```

Determining the CoEdges of a Structure Object

The following code returns the number of CoEdge objects contained by the Structure object 'geom', lists the unique identifiers of the CoEdge objects, and then retrieves them individually.

```
// find how many coEdges are in the Structure object
Lps::Uint32 coEdgeCount;
coEdgeCount = geom->numberOfCoEdges ();
std::cout << "Number of CoEdges: " << coEdgeCount
          << std::endl;

// get unique identifiers of the coEdges in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfCoEdges ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "CoEdge " << *it << std::endl;

// given the list of unique identifiers, retrieve coEdges
// individually and store in a STL vector of CoEdgePtr
Lps::CoEdgePtrList coEdges;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  coEdges.push_back (geom->getCoEdge (*it));
```

Determining the Edges of a Structure Object

The following code returns the number of Edge objects contained by the Structure object 'geom', lists the unique identifiers of the Edge objects, and then retrieves them individually.

```
// find how many edges are in the Structure object
Lps::Uint32 edgeCount;
edgeCount = geom->numberOfEdges ();
std::cout << "Number of Edges: " << edgeCount
```

```
          << std::endl;

// get unique identifiers of the edges in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfEdges ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Edge " << *it << std::endl;

// given the list of unique identifiers, retrieve edges
// individually and store in a STL vector of EdgePtr
Lps::EdgePtrList edges;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  edges.push_back (geom->getEdge (*it));
```

Determining the CoPoints of a Structure Object

The following code returns the number of CoPoint objects contained by the Structure object 'geom', lists the unique identifiers of the CoPoint objects, and then retrieves them individually.

```
// find how many coPoints are in the Structure object
Lps::Uint32 coPointCount;
coPointCount = geom->numberOfCoPoints ();
std::cout << "Number of CoPoints: " << coPointCount
          << std::endl;

// get unique identifiers of the coPoints in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfCoPoints ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "CoPoint " << *it << std::endl;

// given the list of unique identifiers, retrieve coPoints
// individually and store in a STL vector of CoPointPtr
Lps::CoPointPtrList coPoints;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  coPoints.push_back (geom->getCoPoint (*it));
```

Determining the Ppoints of a Structure Object

The following code returns the number of Ppoint objects contained by the Structure object 'geom', lists the unique identifiers of the Ppoint objects, and then retrieves them individually.

```
// find how many ppoints are in the Structure object
Lps::Uint32 ppointCount;
ppointCount = geom->numberOfPpoints ();
std::cout << "Number of Ppoints: " << ppointCount
          << std::endl;

// get unique identifiers of the ppoints in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfPpoints ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Ppoint " << *it << std::endl;

// given the list of unique identifiers, retrieve ppoints
// individually and store in a STL vector of PpointPtr
```

```
Lps::PpointPtrList ppoints;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  ppoints.push_back (geom->getPpoint (*it));
```

Determining the Pcurves of a Structure Object

The following code returns the number of Pcurve objects contained by the Structure object 'geom', lists the unique identifiers of the Pcurve objects, and then retrieves them individually.

```
// find how many pcurves are in the Structure object
Lps::Uint32 pcurveCount;
pcurveCount = geom->numberOfPcurves ();
std::cout << "Number of Pcurves: " << pcurveCount
          << std::endl;

// get unique identifiers of the pcurves in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfPcurves ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Pcurve " << *it << std::endl;

// given the list of unique identifiers, retrieve pcurves
// individually and store in a STL vector of PcurvePtr
Lps::PcurvePtrList pcurves;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  pcurves.push_back (geom->getPcurve (*it));
```

Determining the Surfaces of a Structure Object

The following code returns the number of Surface objects contained by the Structure object 'geom', lists the unique identifiers of the Surface objects, and then retrieves them individually.

```
// find how many surfaces are in the Structure object
Lps::Uint32 surfaceCount;
surfaceCount = geom->numberOfSurfaces ();
std::cout << "Number of Surfaces: " << surfaceCount
          << std::endl;

// get unique identifiers of the surfaces in the
// Structure and print them
Lps::UniqueIdList uidList = geom->getUidsOfSurfaces ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Surface " << *it << std::endl;

// given the list of unique identifiers, retrieve surfaces
// individually and store in a STL vector of SurfacePtr
Lps::SurfacePtrList surfaces;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  surfaces.push_back (geom->getSurface (*it));
```

Determining the Contents of a LEAPS CommonView Object

A LEAPS CommonView is currently composed of CommonView objects and TopologicalView objects. It also has of Property objects,

PropertyGroup objects, Material objects, and MaterialGroup objects. If 'commonView' is a CommonView that has been retrieved from a Structure, the contents of the CommonView object can be queried and retrieved.

Determining the Properties of a CommonView Object

The following code returns the number of Property objects contained by the CommonView object 'commonView', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the CommonView object
Lps::Uint32 propertyCount;
propertyCount = commonView->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the CommonView and
// print them
Lps::UniqueIdList uidList = commonView->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (commonView->getProperty (*it));
```

Determining the PropertyGroups of a CommonView Object

The following code returns the number of PropertyGroup objects contained by the CommonView object 'commonView', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the CommonView object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = commonView->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// CommonView and print them
Lps::UniqueIdList uidList = commonView->getUidsOfPropertyGroups
();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  propertyGroups.push_back (commonView->getPropertyGroup (*it));
```

Determining the Materials of a CommonView Object

The following code returns the number of Material objects contained by the CommonView object 'commonView', lists the unique identifiers of the Material objects, and then retrieves them individually.

```
// find how many materials are in the CommonView object
Lps::Uint32 materialCount;
materialCount = commonView->numberOfMaterials ();
std::cout << "Number of Materials: " << materialCount
          << std::endl;

// get unique identifiers of the materials in the CommonView and
// print them
Lps::UniqueIdList uidList = commonView->getUidsOfMaterials ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Material " << *it << std::endl;

// given the list of unique identifiers, retrieve materials
// individually and store in a STL vector of MaterialPtr
Lps::MaterialPtrList materials;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  materials.push_back (commonView->getMaterial (*it));
```

Determining the MaterialGroups of a CommonView Object

The following code returns the number of MaterialGroup objects contained by the CommonView object 'commonView', lists the unique identifiers of the MaterialGroup objects, and then retrieves them individually.

```
// find how many materialGroups are in the CommonView object
Lps::Uint32 materialGroupCount;
materialGroupCount = commonView->numberOfMaterialGroups ();
std::cout << "Number of MaterialGroups: " << materialGroupCount
          << std::endl;

// get unique identifiers of the materialGroups in the
// CommonView and print them
Lps::UniqueIdList uidList = commonView->getUidsOfMaterialGroups
();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "MaterialGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve materialGroups
// individually and store in a STL vector of MaterialGroupPtr
Lps::MaterialGroupPtrList materialGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  materialGroups.push_back (commonView->getMaterialGroup (*it));
```

Determining the CommonViews of a CommonView Object

The following code returns the number of CommonView objects contained by the CommonView object 'commonView', lists the unique identifiers of the CommonView objects, and then retrieves them individually.

```
// find how many commonViews are in the CommonView object
```

```
Lps::Uint32 commonViewCount;
commonViewCount = commonView->numberOfCommonViews ();
std::cout << "Number of CommonViews: " << commonViewCount
          << std::endl;

// get unique identifiers of the commonViews in the
// CommonView and print them
Lps::UniqueIdList uidList = commonView->getUidsOfCommonViews ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "CommonView " << *it << std::endl;

// given the list of unique identifiers, retrieve commonViews
// individually and store in a STL vector of CommonViewPtr
Lps::CommonViewPtrList commonViews;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  commonViews.push_back (commonView->getCommonView (*it));
```

Determining the TopologicalViews of a CommonView Object

The following code returns the number of TopologicalView objects contained by the CommonView object 'commonView', lists the unique identifiers of the TopologicalView objects, and then retrieves them individually.

```
// find how many topologicalViews are in the CommonView object
Lps::Uint32 topologicalViewCount;
topologicalViewCount = commonView->numberOfTopologicalViews ();
std::cout << "Number of TopologicalViews: "
          << topologicalViewCount << std::endl;

// get unique identifiers of the topologicalViews in the
// CommonView and print them
Lps::UniqueIdList uidList = commonView->getUidsOfTopologicalViews
();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "TopologicalView " << *it << std::endl;

// given the list of unique identifiers, retrieve topologicalViews
// individually and store in a STL vector of TopologicalViewPtr
Lps::TopologicalViewPtrList topologicalViews;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  topologicalViews.push_back
                          (commonView->getTopologicalView (*it));
}
```

Determining the CommonViews that Use the CommonView Object

The following code returns the number of CommonView objects that are used by the CommonView object 'commonView', lists the unique identifiers of these CommonView objects, and then retrieves them individually.

```
// find how many CommonViews that use the CommonView object
Lps::Uint32 viewCount;
```

```
viewCount = commonView->numberOfCommonViewsUsingCommonView ();
std::cout << "Number of CommonViews: " << viewCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// CommonView and print them
Lps::UniqueIdList uidList;
uidList = commonView->getUidsOfCommonViewsUsingCommonView ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "CommonView " << *it << std::endl;

// given the list of unique identifiers, retrieve CommonViews
// individually and store in a STL vector of CommonViewPtr
Lps::CommonViewPtrList views;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  views.push_back
               (commonView->getCommonViewUsingCommonView (*it));
}
```

Determining the Contents of a LEAPS TopologicalView Object

A LEAPS TopologicalView is currently a Solid, Face or Solid that has Property objects, PropertyGroup objects, Material objects, and MaterialGroup objects. If 'topologicalView' is a TopologicalView that has been retrieved from a Structure, the contents of the TopologicalView object can be queried and retrieved.

Determining the Properties of a TopologicalView Object

The following code returns the number of Property objects contained by the TopologicalView object 'topologicalView', lists the unique identifiers of the Property objects, and then retrieves them individually.

```
// find how many properties are in the TopologicalView object
Lps::Uint32 propertyCount;
propertyCount = topologicalView->numberOfProperties ();
std::cout << "Number of Properties: " << propertyCount
          << std::endl;

// get unique identifiers of the properties in the
// TopologicalView and print them
Lps::UniqueIdList uidList;
uidList = topologicalView->getUidsOfProperties ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Property " << *it << std::endl;

// given the list of unique identifiers, retrieve properties
// individually and store in a STL vector of PropertyPtr
Lps::PropertyPtrList properties;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  properties.push_back (topologicalView->getProperty (*it));
```

Determining the PropertyGroups of a TopologicalView Object

The following code returns the number of PropertyGroup objects contained by the TopologicalView object 'topologicalView', lists the unique identifiers of the PropertyGroup objects, and then retrieves them individually.

```
// find how many propertyGroups are in the TopologicalView object
Lps::Uint32 propertyGroupCount;
propertyGroupCount = topologicalView->numberOfPropertyGroups ();
std::cout << "Number of PropertyGroups: " << propertyGroupCount
          << std::endl;

// get unique identifiers of the propertyGroups in the
// TopologicalView and print them
Lps::UniqueIdList uidList = topologicalView-
>getUidsOfPropertyGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "PropertyGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve propertyGroups
// individually and store in a STL vector of PropertyGroupPtr
Lps::PropertyGroupPtrList propertyGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  propertyGroups.push_back
                        (topologicalView->getPropertyGroup (*it));
}
```

Determining the Materials of a TopologicalView Object

The following code returns the number of Material objects contained by the TopologicalView object 'topologicalView', lists the unique identifiers of the Material objects, and then retrieves them individually.

```
// find how many materials are in the TopologicalView object
Lps::Uint32 materialCount;
materialCount = topologicalView->numberOfMaterials ();
std::cout << "Number of Materials: " << materialCount
          << std::endl;

// get unique identifiers of the materials in the
// TopologicalView and print them
Lps::UniqueIdList uidList;
uidList = topologicalView->getUidsOfMaterials ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Material " << *it << std::endl;

// given the list of unique identifiers, retrieve materials
// individually and store in a STL vector of MaterialPtr
Lps::MaterialPtrList materials;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  materials.push_back (topologicalView->getMaterial (*it));
```

Determining the MaterialGroups of a TopologicalView Object

The following code returns the number of MaterialGroup objects contained by the TopologicalView object 'topologicalView', lists the unique identifiers of the MaterialGroup objects, and then retrieves them individually.

```
// find how many materialGroups are in the TopologicalView object
Lps::Uint32 materialGroupCount;
materialGroupCount = topologicalView->numberOfMaterialGroups ();
std::cout << "Number of MaterialGroups: " << materialGroupCount
          << std::endl;

// get unique identifiers of the materialGroups in the
// TopologicalView and print them
Lps::UniqueIdList uidList = topologicalView-
>getUidsOfMaterialGroups ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "MaterialGroup " << *it << std::endl;

// given the list of unique identifiers, retrieve materialGroups
// individually and store in a STL vector of MaterialGroupPtr
Lps::MaterialGroupPtrList materialGroups;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  materialGroups.push_back
                        (topologicalView->getMaterialGroup (*it));
}
```

Determining the Leaps Object Type of a TopologicalView Object

The following code determines whether the TopologicalView object is a Solid, Face, or Solid, retrieves the LEAPS object that represents the TopologicalView, and prints the unique identifiers of both objects.

```
// find type of LEAPS object that represents the
// TopologicalView object
if (topologicalView->objectType () == Lps::SolidObject)
{
  Lps::SolidPtr solid = topologicalView->getSolid ();
  std::cout << "TopologicalView " << topologicalView->uniqueId ()
            << " is Solid " << solid->uniqueId () << std::endl;
}
else if (topologicalView->objectType () == Lps::FaceObject)
{
  Lps::FacePtr face = topologicalView->getFace ();
  std::cout << "TopologicalView " << topologicalView->uniqueId ()
            << " is Face " << face->uniqueId () << std::endl;
}
if (topologicalView->objectType () == Lps::SurfaceObject)
{
  Lps::SurfacePtr surface = topologicalView->getSurface ();
  std::cout << "TopologicalView " << topologicalView->uniqueId ()
            << " is Surface " << surface->uniqueId ()
            << std::endl;
}
```

Determining the CommonViews that use the TopologicalView Object

The following code returns the number of CommonView objects that use the TopologicalView object 'topologicalView', lists the unique identifiers of the CommonView objects, and then retrieves them individually.

```
// find how many CommonViews that use the TopologicalView object
Lps::Uint32 viewCount;
viewCount =
      topologicalView->numberOfCommonViewsUsingTopologicalView ();
std::cout << "Number of CommonViews: " << viewCount << std::endl;

// get unique identifiers of the CommonViews that use the
// TopologicalView and print them
Lps::UniqueIdList uidList;
uidList =
     topologicalView->getUidsOfCommonViewsUsingTopologicalView ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "CommonView " << *it << std::endl;

// given the list of unique identifiers, retrieve CommonViews
// individually and store in a STL vector of CommonViewPtr
Lps::CommonViewPtrList commonViews;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  commonViews.push_back
                      (topologicalView->getTopologicalView (*it));
}
```

Determining the Contents of a LEAPS Solid Object

A LEAPS Solid object is bounded by an OrientedClosedShell object. This object is the outer boundary of the Solid. The Solid object may also have zero or more OrientedClosedShell objects that define the voids in the Solid object. If 'solid' is a Solid that has been retrieved from a Structure, the contents of the Solid object can be queried and retrieved.

Determining the Outershell of a Solid Object

The following code returns the OrientedClosedShell's unique identifier that is the outer boundary of the Solid and then retrieves it.

```
// get unique identifiers of the outershell and print it
std::string uid = solid->getUidOfOuterShell ();
std::cout << "Solid " << solid->uniqueId ()
          << "has OrientedClosedShell " << uid
          << "as an outer shell" << std::endl;

// retrieve the outer shell of the Solid
Lps::OrientedClosedShellPtr outerShell;
outerShell = solid->getOuterShell ();
```

Determining the Voids of a Solid Object

The following code returns the number of OrientedClosedShell objects that are voids in the Solid object 'solid', lists the unique identifiers of these OrientedClosedShell objects, and then retrieves them individually.

```
// find how many voids are in the Solid object
Lps::Uint32 voidCount;
voidCount = solid->numberOfVoidShells ();
std::cout << "Number of Void Shells: " << voidCount
          << std::endl;

// get unique identifiers of the void shells in the Solid
// and print them
Lps::UniqueIdList uidList = solid->getUidsOfVoidShells ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Void Shell " << *it << std::endl;

// given the list of unique identifiers, retrieve void shells
// individually and store in a STL vector of
// OrientedClosedShellPtr
Lps::OrientedClosedShellPtrList voidShells;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  voidShells.push_back (solid->getVoidShell (*it));
```

Determining the TopologicalView that Represents the Solid Object

If the Solid object, 'solid,' is represented by a TopologicalView, the following code returns that TopologicalView object.

```
// if TopologicalView exists, retrieve that TopologicalView
// and print its unique id
if (solid->doesTopologicalViewExist ())
{
  Lps::TopologicalViewPtr view;
  view = solid->getTopologicalView ();
  std::cout << "TopologicalView " << view->uniqueId ()
            << " represents Solid " << solid->uniqueId ()
            << std::endl;
}
```

Determining the Contents of a LEAPS OrientedClosedShell Object

A LEAPS OrientedClosedShell object is bounded by one or more Face objects that form a closed shell. The OrientedClosedShell object is oriented such that all face normals are either pointing inward or outward. If 'shell' is an OrientedClosedShell that has been retrieved from a Structure, the contents of the OrientedClosedShell object can be queried and retrieved.

Determining the Orientation of an OrientedClosedShell Object

The following code determines the orientation of the OrientedClosedShell object 'shell.'

```
// determine orientation of OrientedClosedShell and print it
Lps::OrientationEnum oriented;
oriented = shell->orientation ();
if (oriented == Lps::OutwardOrientation)
  std::cout << "Orientation is outward." << std::endl;
else if (oriented == Lps::InwardOrientation)
  std::cout << "Orientation is inward." << std::endl;
else
  std::cout << "Orientation is unknown." << std::endl;
```

Determining the Faces of an OrientedClosedShell Object

The following code returns the number of Face objects that compose the OrientedClosedShell object 'shell', lists the unique identifiers of these Face objects, and then retrieves them individually.

```
// find how many faces are in the OrientedClosedShell object
Lps::Uint32 faceCount;
faceCount = shell->numberOfFaces ();
std::cout << "Number of Faces: " << faceCount
          << std::endl;

// get unique identifiers of the faces in the
// OrientedClosedShell and print them
Lps::UniqueIdList uidList = shell->getUidsOfFaces ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Face " << *it << std::endl;

// given the list of unique identifiers, retrieve faces
// individually and store in a STL vector of FacePtr
Lps::FacePtrList faces;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  faces.push_back (shell->getFace (*it));
```

Determining the Solids that Use the OrientedClosedShell Object

If the OrientedClosedShell object, 'shell,' is used by a Solid, the following code lists the unique identifiers of the Solid objects that use 'shell,' and then retrieves them individually.

```
// find how many solids are used by the OrientedClosedShell object
Lps::Uint32 solidCount;
solidCount = shell->numberOfSolidsUsingOrientedClosedShell ();
std::cout << "Number of Solids Using OrientedClosedShell: "
          << solidCount << std::endl;

// get unique identifiers of the solids used by the
// OrientedClosedShell and print them
Lps::UniqueIdList uidList;
uidList = shell->getUidsOfSolidsUsingOrientedClosedShell ();
Lps::UniqueIdList::iterator it;
```

```
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  std::cout << "Solid Used By OrientedClosedShell: " << *it
            << std::endl;
}

// given the list of unique identifiers, retrieve solids
// individually and store in a STL vector of SolidPtr
Lps::SolidPtrList solids;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  solids.push_back
                  (shell->getSolidUsingOrientedClosedShell (*it));
}
```

Determining the Contents of a LEAPS Face Object

A LEAPS Face object is bounded by an EdgeLoop object. This object is the outer boundary of the Face. The Face object may also have zero or more EdgeLoop objects that define the holes in the Face object. If 'face' is a Face that has been retrieved from a Structure, the contents of the Face object can be queried and retrieved.

Determining the Orientation of an Face Object

The following code determines the orientation of the Face object 'face.'

```
// determine orientation of Face and print it
Lps::OrientationEnum oriented;
oriented = face->orientation ();
if (oriented == Lps::OutwardOrientation)
  std::cout << "Orientation is outward." << std::endl;
else if (oriented == Lps::InwardOrientation)
  std::cout << "Orientation is inward." << std::endl;
else
  std::cout << "Orientation is unknown." << std::endl;
```

Determining the Outer Loop of a Face Object

The following code returns the EdgeLoop's unique identifier that is the outer boundary of the Face and then retrieves it.

```
// get unique identifiers of the outerloop and print it
std::string uid = face->getUidOfOuterLoop ();
std::cout << "Solid " << solid->uniqueId ()
          << "has EdgeLoop " << uid
          << "as an outer loop" << std::endl;

// retrieve the outer loop of the face
Lps::EdgeLoopPtr outerLoop;
outerLoop = face->getOuterLoop ();
```

Determining the Inner Loops of a Face Object

The following code returns the number of EdgeLoop objects that represent holes in the Face object 'face', lists the unique identifiers of these EdgeLoop objects, and then retrieves them individually.

```
// find how many edgeLoops are in the Face object
```

```
Lps::Uint32 edgeLoopCount;
edgeLoopCount = face->numberOfInnerLoops ();
std::cout << "Number of Inner Loops: " << edgeLoopCount
          << std::endl;

// get unique identifiers of the edgeLoops in the
// Face and print them
Lps::UniqueIdList uidList = face->getUidsOfInnerLoops ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "EdgeLoop " << *it << std::endl;

// given the list of unique identifiers, retrieve edgeLoops
// individually and store in a STL vector of EdgeLoopPtr
Lps::EdgeLoopPtrList edgeLoops;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  edgeLoops.push_back (face->getInnerLoop (*it));
```

Determining the OrientedClosedShells that Use the Face Object

If the Face object, 'face,' is used by an OrientedClosedShell, the following code lists the unique identifiers of the OrientedClosedShell objects that use 'face,' and then retrieves them individually.

```
// find how many shells are used by the Face object
Lps::Uint32 shellCount;
shellCount = face->numberOfOrientedClosedShellsUsingFace ();
std::cout << "Number of OrientedClosedShells Using Face: "
          << shellCount << std::endl;

// get unique identifiers of the shells used by the
// Face and print them
Lps::UniqueIdList uidList;
uidList = face->getUidsOfOrientedClosedShellsUsingFace ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  std::cout << "OrientedClosedShell Used By Face: " << *it
            << std::endl;
}

// given the list of unique identifiers, retrieve shells
// individually and store in a STL vector of
// OrientedClosedShellPtr
Lps::OrientedClosedShellPtrList shells;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  shells.push_back (face->getOrientedClosedShellUsingFace (*it));
```

Determining the TopologicalView that Represents the Face Object

If the Face object, 'face,' is represented by a TopologicalView, the following code returns that TopologicalView object.

```
// if TopologicalView exists, retrieve that TopologicalView
// and print its unique id
if (face->doesTopologicalViewExist ())
{
  Lps::TopologicalViewPtr view;
```

```
  view = face->getTopologicalView ();
  std::cout << "TopologicalView " << view->uniqueId ()
            << " represents Face " << face->uniqueId ()
            << std::endl;
}
```

Determining the Surface the Face Object Is On

If the Face object, 'face,' the Surface object that the face in on can be retrieved by the following code.

```
// retrieve surface that face lies on and print its unique id
Lps::SurfacePtr surface = face->getSurface ();
std::cout << "Face " << face->uniqueId ()
          << " lies on Surface " << surface->uniqueId ()
          << std::endl;
```

Determining the Contents of a LEAPS EdgeLoop Object

A LEAPS EdgeLoop object is composed of one or more Edge objects that form a closed loop. The EdgeLoop object is oriented such that the loop is either counter clockwise or clockwise. If 'edgeLoop' is an EdgeLoop that has been retrieved from a Structure, the contents of the EdgeLoop object can be queried and retrieved.

Determining the Orientation of an EdgeLoop Object

The following code determines the orientation of the EdgeLoop object 'edgeLoop.'

```
// determine orientation of EdgeLoop and print it
Lps::OrientationEnum oriented;
oriented = edgeLoop->orientation ();
if (oriented == Lps::CounterClockwiseOrientation)
  std::cout << "Orientation is counter clockwise." << std::endl;
else if (oriented == Lps::ClockwiseOrientation)
  std::cout << "Orientation is clockwise." << std::endl;
else
  std::cout << "Orientation is unknown." << std::endl;
```

Determining the Edges of an EdgeLoop Object

The following code returns the number of Edge objects that compose the EdgeLoop object 'edgeLoop', lists the unique identifiers of these Edge objects, and then retrieves them individually.

```
// find how many edges are in the EdgeLoop object
Lps::Uint32 edgeCount;
edgeCount = edgeLoop->numberOfEdges ();
std::cout << "Number of Edges: " << edgeCount
          << std::endl;

// get unique identifiers of the edges in the
// EdgeLoop and print them
Lps::UniqueIdList uidList = edgeLoop->getUidsOfEdges ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Edge " << *it << std::endl;
```

```
// given the list of unique identifiers, retrieve edges
// individually and store in a STL vector of EdgePtr
Lps::EdgePtrList edges;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  edges.push_back (edgeLoop->getEdge (*it));
```

Determining the Faces that Use the EdgeLoop Object

If the EdgeLoop object, 'edgeLoop,' is used by a Face, the following code lists the unique identifiers of the Face objects that use 'edgeLoop,' and then retrieves them individually.

```
// find how many faces are used by the EdgeLoop object
Lps::Uint32 faceCount;
faceCount = edgeLoop->numberOfFacesUsingEdgeLoop ();
std::cout << "Number of Faces Using EdgeLoop: "
          << faceCount << std::endl;

// get unique identifiers of the faces used by the
// EdgeLoop and print them
Lps::UniqueIdList uidList;
uidList = edgeLoop->getUidsOfFacesUsingEdgeLoop ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
{
  std::cout << "Face Used By EdgeLoop: " << *it
            << std::endl;
}

// given the list of unique identifiers, retrieve faces
// individually and store in a STL vector of FacePtr
Lps::FacePtrList faces;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  faces.push_back (edgeLoop->getFaceUsingEdgeLoop (*it));
```

Determining the Contents of a LEAPS Edge Object

A LEAPS Edge object is an oriented segment of a Pcurve object. It is defined by a start Ppoint object and an end Ppoint object. If 'edge' is an Edge that has been retrieved from a Structure, the contents of the Edge object can be queried and retrieved.

Determining the Start Point of an Edge Object

For the Edge object 'edge', the following code lists the unique identifier of the Ppoint object that starts 'edge,' and then retrieve this Ppoint.

```
// get unique identifier of the start Ppoint of the edge
// and print it
std::string uid = edge->getUidOfStartPoint ();
std::cout << "Ppoint " << *it << " starts Edge "
          << edge->uniqueId () << std::endl;

// retrieve start Ppoint object for edge
Lps::PpointPtr startPt = edge->getStartPoint ();
```

Determining the End Point of an Edge Object

For the Edge object 'edge', the following code lists the unique identifier of the Ppoint object that ends 'edge,' and then retrieve this Ppoint.

```
// get unique identifier of the end Ppoint of the edge
// and print it
std::string uid = edge->getUidOfEndPoint ();
std::cout << "Ppoint " << *it << " ends Edge "
          << edge->uniqueId () << std::endl;

// retrieve end Ppoint object for edge
Lps::PpointPtr endPt = edge->getEndPoint ();
```

Determining the Pcurve that the Edge Object Lies on

For the Edge object 'edge', the following lists the unique identifier of the Pcurve object the edge lies on, and then retrieve it.

```
// get unique identifier of the Pcurve of the edge
// and print it
std::string uid = edge->getUidOfPcurve ();
std::cout << "Edge " << edge->uniqueId () << " lies on Pcurve "
          << *it << std::endl;

// retrieve Pcurve object for edge
Lps::PcurvePtr pcrv = edge->getPcurve ();
```

Determining the Surface that the Edge Object Lies on

For the Edge object 'edge', the following lists the unique identifier of the Surface object the edge lies on, and then retrieve it.

```
// get unique identifier of the Surface of the edge
// and print it
std::string uid = edge->getUidOfSurface ();
std::cout << "Edge " << edge->uniqueId () << " lies on Surface "
          << *it << std::endl;

// retrieve Surface object for edge
Lps::SurfacePtr surf = edge->getSurface ();
```

Determining the CoEdge that the Edge Object Is A Part Of

If the Edge object, 'edge,' is part of a CoEdge object, the following code list the unique identifier that CoEdge object and retrieves it.

```
// get unique identifier of CoEdge of the edge
// and print it
std::string uid = edge->getUidOfCoEdge ();
if (uid.length () == 0)
{
  std::cout << "Edge " << edge->uniqueId ()
            << " is Not part of CoEdge." << std::endl;
}
else
{
  std::cout << "Edge " << edge->uniqueId ()
            << " is part of CoEdge  " << uid << std::endl;
  // retrieve CoEdge object for edge
```

```
  Lps::CoEdgePtr coEdge = edge->getCoEdge ();
}
```

Determining the EdgeLoops that Use the Edge Object

If the Edge object, 'edge,' is used by an EdgeLoop, the following code lists the unique identifiers of the EdgeLoop objects that use 'edge,' and then retrieves them individually.

```
// find how many edgeLoops are used by the Edge object
Lps::Uint32 edgeLoopCount;
edgeLoopCount = edgeLoop->numberOfEdgeLoopsUsingEdge ();
std::cout << "Number of EdgeLoops Using Edge: "
          << edgeLoopCount << std::endl;

// get unique identifiers of the edgeLoops used by the
// Edge and print them
Lps::UniqueIdList uidList;
uidList = edgeLoop->getUidsOfEdgeLoopsUsingEdge ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "EdgeLoop Used By Edge: " << *it << std::endl;

// given the list of unique identifiers, retrieve edgeLoops
// individually and store in a STL vector of EdgeLoopPtr
Lps::EdgeLoopPtrList edgeLoops;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  edgeLoops.push_back (edgeLoop->getEdgeLoopUsingEdge (*it));
```

Determining the Contents of a LEAPS CoEdge Object

A LEAPS CoEdge object is composed of two or more Edge objects that are logically coincident. If 'coEdge' is an CoEdge that has been retrieved from a Structure, the contents of the CoEdge object can be queried and retrieved.

Determining the Edges of a CoEdge Object

The following code returns the number of Edge objects that compose the CoEdge object 'coEdge', lists the unique identifiers of these Edge objects, and then retrieves them individually.

```
// find how many edges are in the CoEdge object
Lps::Uint32 edgeCount;
edgeCount = coEdge->numberOfEdges ();
std::cout << "Number of Edges: " << edgeCount
          << std::endl;

// get unique identifiers of the edges in the
// CoEdge and print them
Lps::UniqueIdList uidList = coEdge->getUidsOfEdges ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Edge " << *it << std::endl;

// given the list of unique identifiers, retrieve edges
// individually and store in a STL vector of EdgePtr
Lps::EdgePtrList edges;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  edges.push_back (coEdge->getEdge (*it));
```

Determining the Contents of a LEAPS Ppoint Object

A LEAPS Ppoint object is a parametric point on a *Pcurve* object. A Ppoint may start zero, one, or two Edge objects and end zero, one, or two Edge objects. Additionally, a Ppoint object may be part of a CoPoint object. If 'ppoint' is a Ppoint that has been retrieved from a Structure, the contents of the Ppoint object can be queried and retrieved.

Determining the Edges the Ppoint Object Starts and Ends

The following code illustrates how to retrieve the Edges that the Ppoint object 'ppoint' starts and ends.

```
// find how many edges the Ppoint object starts
Lps::Uint32 startEdgeCount;
startEdgeCount = ppoint->numberOfEdgesIStart ();
std::cout << "Ppoint " << ppoint->uniqueId () << " starts "
          << startEdgeCount << " Edges." << std::endl;

// find how many edges the Ppoint object ends
Lps::Uint32 endEdgeCount;
endEdgeCount = ppoint->numberOfEdgesIEnd ();
std::cout << "Ppoint " << ppoint->uniqueId () << " ends "
          << endEdgeCount << " Edges." << std::endl;

// get unique identifiers of the edges the Ppoint starts
Lps::UniqueIdList uidList = ppoint->getUidsOfEdgesIStart ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Ppoint " << ppoint->uniqueId () << " starts "
            << " Edge " << *it << std::endl;

// get unique identifiers of the edges the Ppoint ends
uidList = ppoint->getUidsOfEdgesIEnd ();
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Ppoint " << ppoint->uniqueId () << " ends "
            << " Edge " << *it << std::endl;

// retrieve edges in a STL vector of EdgePtr that
// the Ppoint starts
Lps::EdgePtrList startEdges = ppoint->getEdgesIStart ();

// retrieve edges in a STL vector of EdgePtr that
// the Ppoint ends
Lps::EdgePtrList endEdges = ppoint->getEdgesIEnd ();
```

Determining the Pcurve Object that the Ppoint Object Lies on

The following code illustrates how to retrieve the Pcurve object that the Ppoint object 'ppoint' lies on.

```
// find Pcurve object that the Ppoint object lies on
Lps::PcurvePtr pcrv = ppoint->getPcurve ();
std::cout << "Ppoint " << ppoint->uniqueId ()
          << " lies on Pcurve " << pcrv->uniqueId () << std::endl;
```

Determining the location of the Ppoint Object

The following code illustrates how to retrieve the location (i.e. the parametric value) of the Ppoint object 'ppoint.'.

```
// find the location of the Ppoint object on the Pcurve
Lps::Real64 loc = ppoint->location ();
std::cout << "Ppoint " << ppoint->uniqueId () << " is located at "
          << loc << std::endl;
```

The following code illustrates how to retrieve the cartesian location of the Ppoint object 'ppoint.'

```
// find the cartesian location of the Ppoint object on the Pcurve
Lps::CartesianLocation cartesianLoc;
cartesianLoc = ppoint->evalForCartesianLoc ();
std::cout << "Ppoint " << ppoint->uniqueId ()
          << " is located at (" << cartesianLoc.x() << ", "
          << cartesianLoc.y() << ", "
          << cartesianLoc.z() << ")" << std::endl;
```

The following code illustrates how to retrieve the pcurve location of the Ppoint object 'ppoint.'

```
// find the pcurve location of the Ppoint object on the Pcurve
Lps::PcurveLocation pcurveLoc;
pcurveLoc = ppoint->evalForPcurveLoc ();
std::cout << "Ppoint " << ppoint->uniqueId ()
          << " is located at s = " << pcurveLoc.s() << ", u = "
          << pcurveLoc.u() << ", v = " << pcurveLoc.v()<< ", x = "
          << pcurveLoc.x() <<", y = " << pcurveLoc.y()
          <<", z = " << pcurveLoc.z () << std::endl;
```

Determining the CoPoint that the Ppoint Object is a Part of

If the Ppoint object, 'ppoint,' is part of a CoPoint object, the following code list the unique identifier that CoPoint object and retrieves it.

```
// get unique identifier of CoPoint of the ppoint
// and print it
std::string uid = ppoint->getUidOfCoPoint ();
if (uid.length () == 0)  // no CoPoint
{
  std::cout << "Ppoint " << ppoint->uniqueId ()
            << " is Not part of CoPoint." << std::endl;
}
else
{
  std::cout << "Ppoint " << ppoint->uniqueId ()
            << " is part of CoPoint  " << uid << std::endl;
  // retrieve CoPoint object for ppoint
  Lps::CoPointPtr coPoint = ppoint->getCoPoint ();
}
```

Determining the Contents of a LEAPS CoPoint Object

A LEAPS CoPoint object is composed of two or more Ppoint objects that are logically coincident. If 'coPoint' is an CoPoint that has been retrieved

from a Structure, the contents of the CoPoint object can be queried and retrieved.

Determining the Ppoints of a CoPoint Object

The following code returns the number of Ppoint objects that compose the CoPoint object 'coPoint', lists the unique identifiers of these Ppoint objects, and then retrieves them individually.

```
// find how many Ppoints are in the CoPoint object
Lps::Uint32 ppointCount;
ppointCount = coPoint->numberOfPpoints ();
std::cout << "Number of Ppoints: " << ppointCount
          << std::endl;

// get unique identifiers of the Ppoints in the
// CoPoint and print them
Lps::UniqueIdList uidList = coPoint->getUidsOfPpoints ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Ppoint " << *it << std::endl;

// given the list of unique identifiers, retrieve Ppoints
// individually and store in a STL vector of PpointPtr
Lps::PpointPtrList ppoints;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  ppoints.push_back (coPoint->getPpoint (*it));
```

Determining the Cartesian location of the CoPoint Object

The following code illustrates how to retrieve the cartesian location of the CoPoint object 'coPoint.'

```
// find the cartesian location of the CoPoint object
Lps::CartesianLocation cartesianLoc;
cartesianLoc = coPoint->location ();
std::cout << "CoPoint " << coPoint->uniqueId ()
          << " is located at (" << cartesianLoc.x() << ", "
          << cartesianLoc.y() << ", "
          << cartesianLoc.z() << ")" << std::endl;
```

Determining the Contents of a LEAPS Pcurve Object

A LEAPS Pcurve object is a parametric spline curve on a *Surface* object. The Pcurve object may also have zero or more Ppoint objects that are mapped to it. If 'pcurve' is a Pcurve that has been retrieved from a Structure, the contents of the Pcurve object can be queried and retrieved.

Determining the Surface that the Pcurve Object is Mapped to

The following code returns the Surface's unique identifier that the Pcurve object is mapped to and then retrieves it.

```
// get unique identifiers of the outershell and print it
std::string uid = pcurve->getUidOfSurface ();
std::cout << "Pcurve " << pcurve->uniqueId ()
          << "is mapped to Surface " << uid << std::endl;
```

```
// retrieve the Surface that the Pcurveis mapped to
Lps::SurfacePtr surface;
surface = pcurve->getSurface ();
```

Determining the Ppoints that are Mapped to a Pcurve Object

The following code returns the number of Ppoint objects that is mapped to the Pcurve object 'pcurve', lists the unique identifiers of these Ppoint objects, and then retrieves them individually.

```
// find how many Ppoints that are mapped to the Pcurve object
Lps::Uint32 ppointCount;
ppointCount = pcurve->numberOfMappedPpoints ();
std::cout << "Number of Mapped Ppoints: " << ppointCount
          << std::endl;

// get unique identifiers of the Ppoints that are mapped
// to the Pcurve object and print them
Lps::UniqueIdList uidList = pcurve->getUidsOfMappedPpoints ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Ppoint " << *it << std::endl;

// given the list of unique identifiers, retrieve Ppoints
// individually and store in a STL vector of PpointPtr
Lps::PpointPtrList ppoints;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  ppoints.push_back (pcurve->getMappedPpoint (*it));
```

Determining the Contents of a LEAPS Surface Object

A LEAPS Surface object is a non-uniform rational b-spline representation of a surface in Cartesian space. The Surface object may also have zero or more Pcurve objects that are mapped to it. If 'surface' is a Surface that has been retrieved from a Structure, the contents of the Surface object can be queried and retrieved.

Determining the Pcurves that are Mapped to a Surface Object

The following code returns the number of Pcurve objects that is mapped to the Surface object 'surface', lists the unique identifiers of these Pcurve objects, and then retrieves them individually.

```
// find how many Pcurves that are mapped to the Surface object
Lps::Uint32 pcurveCount;
pcurveCount = surface->numberOfMappedPcurves ();
std::cout << "Number of Mapped Pcurves: " << pcurveCount
          << std::endl;

// get unique identifiers of the Pcurves that are mapped
// to the Surface object and print them
Lps::UniqueIdList uidList = surface->getUidsOfMappedPcurves ();
Lps::UniqueIdList::iterator it;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  std::cout << "Pcurve " << *it << std::endl;
```

```
// given the list of unique identifiers, retrieve Pcurves
// individually and store in a STL vector of PcurvePtr
Lps::PcurvePtrList pcurves;
for (it = uidList.begin () ; it != uidList.end () ; ++it)
  pcurves.push_back (surface->getMappedPcurve (*it));
```

Determining the TopologicalView that Represents the Surface Object

If the Surface object, 'surface,' is represented by a TopologicalView, the following code returns that TopologicalView object.

```
// if TopologicalView exists, retrieve that TopologicalView
// and print its unique id
if (surface->doesTopologicalViewExist ())
{
  Lps::TopologicalViewPtr view;
  view = surface->getTopologicalView ();
  std::cout << "TopologicalView " << view->uniqueId ()
            << " represents Surface " << surface->uniqueId ()
            << std::endl;
}
```